Toward a Framework for Capturing and Using Architecture Design Knowledge

Muhammad Ali Babar, Ian Gorton, and Ross Jeffery
School of Computer Science & Engineering, University of New South Wales, Australia and National ICT Australia Ltd.
{malibaba, iango, rossj}@cse.unsw.edu.au

UNSW-CSE-TR- 0513

June 2005

THE UNIVERSITY OF
NEW SOUTH WALES

Abstract

Management of architecture knowledge is vital for improving an organization’s architectural capabilities. Despite the recognition of the importance of capturing and reusing architecture knowledge, there is no suitable support mechanism. We propose a conceptual framework for providing appropriate guidance and tool support for making tacit or informally described architecture knowledge explicit. This framework identifies different approaches to capturing implicit architecture knowledge. We discuss different usages of the captured knowledge to improve the effectiveness of architecting process. The report also presents a brief description of a prototype of a web-based architecture knowledge management tool to support the storage and retrieval of the captured knowledge. The report concludes with open issues that we plan to address in order to successfully transfer this support mechanism for capturing and using architecture knowledge to the industry.

1. Introduction

Software Architecture (SA) design and evaluation involves complex and knowledge intensive tasks [36, 42]. The complexity lies in the fact that tradeoffs need to be made to satisfy current and future requirements of a potentially large set of stakeholders, who may have competing vested interests in architectural decisions[2, 20]. The knowledge required to make suitable architectural choices is broad, complex, and evolving, and can be beyond the capabilities of any single architect.

Due to the recognition of the importance and far reaching influence of the architectural decisions, several approaches (such as Architecture Tradeoff Analysis Method (ATAM) [15], 4+1 views [29], Rationale Unified Process (RUP) [28] and architecture-based development [10]) have been developed to support architecting process. While these approaches help manage complexity by using systematic approaches to reason about various design decisions, they provide very little guidance or support to capture and maintain the details on which design decisions are based, along with explanations of the use of certain types of design constructs (such as patterns, styles, tactics and others). Such information represents architecturally significant knowledge, which can be valuable throughout the software development lifecycle [12, 18].

Lack of a systematic approach to capture and use architecture knowledge may preclude organizations from growing their architecting capability and reusing architectural assets. Moreover, the knowledge concerning the domain analysis, patterns used, design options evaluated and design decisions made is implicitly embedded in the architecture and/or becomes tacit knowledge of the architect [12, 42, 44].

Apart from architectural artifacts created during architecting activities, there are several other sources of architecture knowledge. These include architecture styles and patterns [11, 13, 40], design patterns [19], architecture and design tactics [7, 11]. While these sources are aimed at explicitly codifying different types of architecture knowledge, some vital pieces of knowledge are either omitted or informally described. For instance, many pattern documentation formats do not explicitly describe the "*forces*[1]" of a pattern. We have also found that each pattern's documentation informally describes the schemas of synergistic relationships among patterns, quality attributes and scenarios. These can be captured as reusable artifacts in a format that provides architectural knowledge at a level of abstraction appropriate for the architecture design phase [3, 5].

Our research is aimed at improving the quality of architecting process. This is achieved by developing effective knowledge management structures to facilitate the capture and management of implicit architecture knowledge generated during architecting activities or informally described in sources such as [7, 11, 13, 19]. We have been developing a support mechanism to facilitate the capture and use of architecture knowledge by using concepts from knowledge management [35, 37], experience factories [8, 9], and pattern-mining [5, 47] paradigms.

This report presents a conceptual framework for capturing implicit architecture knowledge as reusable artifacts and managing it with a knowledge repository. This makes such knowledge readily available to improve architecture-based software development process.

[1] *The forces of a pattern describe the factors which can cause a problem if they interfere with one another. A pattern attempts to resolve clashes among those factors. Discussion of forces also captures tradeoffs in a pattern.*

The framework identifies various approaches to capture implicit and explicit design and process knowledge during architecting process, along with an approach to distil and document architecture knowledge from patterns. The novelty of the approach resides in its ability to incorporate all the components into an integrated approach, which has been incrementally implemented in a web-based tool.
The reminder of this report is organized as follows. In Section 2 we describe the theoretical background and motivation that stimulated our research in software architecture knowledge management. Section 3 presents a conceptual framework for capturing architecture knowledge. Section 4 describes usages of the captured knowledge. A brief description of a prototype tool is given in Section 5 and Section 6 concludes the report.

2. Theoretical Background and Motivation

In this section we briefly discuss the theoretical concepts that underpin our approach to manage architecture knowledge for supporting and improving architecture processes.

2.1. Architecture-Based Software Development

Software architecture embodies some of earliest design decisions, which are hard and expensive to change if found flawed during downstream development activities. Since the quality attributes (such as maintainability, reliability) of complex software systems largely depend on the overall software architecture of such systems [11], a systematic and integrated approach is required to address architectural issues throughout the software development lifecycle; such approach is called architecture-based development [10]. One of the main characteristics of architecture-based development is the role of quality attributes and architecture styles and patterns, which provide the basis for the design and evaluation of architectural decisions in this development approach [33]. Figure 1 shows a high level process model of architecture-based development that consists of six steps, each having several activities and tasks.

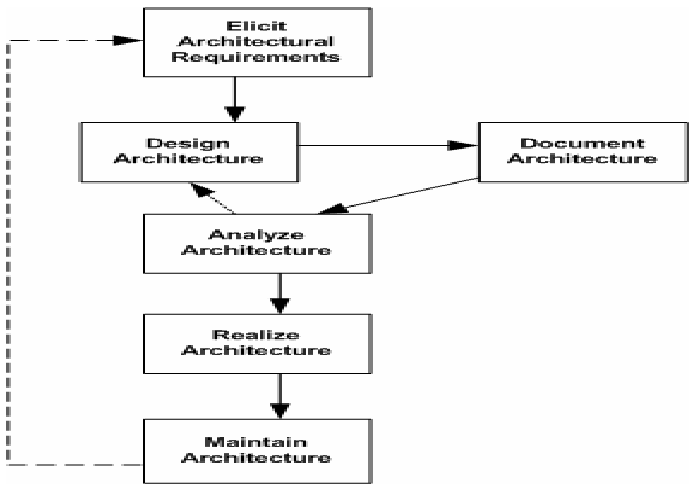


Figure 1: Architecture-Based Development Process model [10].

Architectural requirements are those requirements that have broad cross-functional implications. Such requirements are usually elicited and specified using quality sensitive scenarios [11]. Scenarios have been used for a long time in several areas of different disciplines (military and business strategy, decision making,). The software engineering community started using scenarios in user-interface engineering, requirements elicitation, performance modelling and more recently in SA evaluation [27]. Scenarios are quite effective for specifying architectural requirements because they are very flexible. Scenarios can be used to characterize most of the architectural requirements. For example, we can use scenarios that represent failure to examine availability and reliability, scenarios that represent change requests to analyze modifiability, scenarios that represent threats to analyze security, or scenarios that represent ease of use to analyze usability. Moreover, scenarios are normally concrete, enabling the user to understand their detailed effect [31].

Architecture design is an iterative process, making incremental decisions to satisfy functional and architectural requirements. Architecture design decisions are mainly motivated by architecture requirements, which provide the criteria used to reason about and justify the architectural choices [10]. Architects usually enlist several design options, which may have the potential of satisfying different non-functional requirements. Then, a selection is made from the available design options in order to satisfy all or most of the desired non-functional requirements. This selection process involves several tradeoff decisions.

Architecture design decisions needs to be documented to support the subsequent design and development decisions. Architecture is documented in terms of views, each view addressing a different perspective of the architecture. Architecture design, documentation and analysis are iterative steps in the process [10]. Having designed and analyzed a suitable architecture, it is realized to create the system, and the architecture is maintained to ensure that the detailed design and implementation decision conform to the original architectural decisions and rationales. Moreover, a modification request that can have architectural implications may results in the continuation of architecture-based development cycle starting with eliciting architectural requirements. Later in the paper, we briefly describe what type of knowledge can be captured or used by each step.

2.2. Knowledge Management Issues in Architecting Process

The architecting process aims to solve a mix of ill- and well-defined problems, which involve processing a significant amount of knowledge. Architects require topic knowledge (learned from text books and courses) and episodic knowledge (experience with the knowledge) [36]. One of the main problems in architecture processes is the lack of capture and access to knowledge underpinning the design decisions and the processes leading to those decisions [4, 12]. This type of knowledge involves things like the impact of certain middleware choices on communication mechanisms between different tiers, why an API is used instead of a wrapper, and who to contact to discuss the performance of different architectural choices.

Much of this knowledge is episodic and usually not documented [42]. The absence of a disciplined approach to capture and maintain architecture knowledge has many downstream consequences. These include:

- the evolution of the system becomes complex and cumbersome, resulting in violation of the fundamental design decisions
- inability to identify design errors

- inadequate clarification of arguments and information sharing about the design artifacts and process,

All these cause loss of substantial knowledge generated during architecture process, thus depriving organizations of a valuable resource, loss of key personnel may mean loss of knowledge [22, 25, 42].

The SA community has developed several methods (such as ATAM [15], PASA [46]) to support a disciplined approach to architectural practices. Some of these do emphasize the need for knowledge management to improve reusability and grow organizational capabilities in the architecture domain. Except for [14], there is no approach that explicitly states what type of knowledge needs to be managed and how, when, where, or by whom. Also, none of the current approaches provides any conceptual framework to design, develop and maintain an appropriate repository of architecture knowledge. Hence we posit that the lack of suitable techniques, tools, and guidance is why architecture design knowledge is not captured.

The software engineering community has been discovering and documenting architecture knowledge accumulated by experienced researchers and practitioners in the forms of architecture or design patterns [13, 19]. These patterns attempts to codify implicit knowledge. However, we have found that the amount of information provided and the level of abstraction used may not be appropriate for the architecture stage – too much detail is counter-productive as expert designers usually follow breadth-first approach [36]. Moreover, we have found that the existing formats of pattern documentation are not appropriate for explicating the schemas of the relationships among scenario, quality attributes, and patterns in a way that makes this knowledge readily reusable. This results in little use/reuse of the architectural artifacts (such as scenarios, quality attributes and tactics) informally described in patterns' documentation [3, 5].

Like any other activity of software development, KM in architecture processes also suffers from other problems such as lack of motivation, resources, lackluster sponsorship by the management [16, 43]. However, these issues are not within the main focus of this paper.

2.3. Architecture Knowledge Management Building Blocks

The major objective of Knowledge Management (KM) is to improve business processes and practices by utilizing individual and organizational knowledge resources. These include skills, capabilities, experiences, routines, cultural norms, and technologies [35]. Software engineering processes need or generate both explicit and implicit knowledge. These are mutually complementary entities that interact with each other in creative activities [34].

KM does not ignore the value or need to address other software development aspects, such as process and technology, nor does it seek to replace them. Instead, it works toward software process improvement by explicitly and systematically addressing the management of knowledge. This includes its acquisition, structuring, storage and effective maintenance [37]. There are two main strategies to manage knowledge:

1. codification or centralization: making tacit knowledge explicit
2. personalization or P2P: supporting knowledge sharing by describing who knows what.

Organizations apply both codification and personalization strategies: one of them in a primary and the other in a secondary role [23]. A hybrid approach to manage knowledge is considered an effective and efficient mechanism of maximizing the benefits of codification and personalization strategies of knowledge management for distributed projects [17].

We posit that architecture knowledge management is a management task, which can be described using the knowledge management task model presented in [20]. This model (Figure 2) consists of two strategic and six operational knowledge management tasks, called the building blocks of KM. These represent activities directly related to knowledge. This model presents an integrated approach to KM and ignoring one or more of the building blocks can interrupt the knowledge cycle [35]. For example, if contextual information about designing an artifact in a particular way is not preserved, it may disappear from organizational or individual memory, making reusability of that artifact difficult.

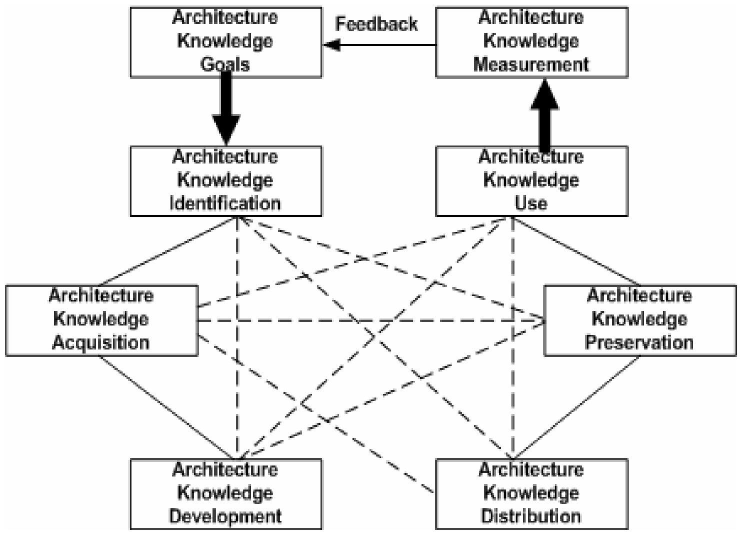

Architecture Knowledge Goals
Feedback
Architecture Knowledge Measurement
Architecture Knowledge Identification
Architecture Knowledge Use
Architecture Knowledge Acquisition
Architecture Knowledge Preservation
Architecture Knowledge Development
Architecture Knowledge Distribution

Figure 2: Building blocks of architecture knowledge management (Modified from [35])

Architecture knowledge goals, a strategic task, describe the objectives of managing knowledge and the expected benefits. For instance, improve quality of architecture decisions, reusability of architectural artifacts, architecture maintenance and evolution, and others. Architecture knowledge measurement is another strategic level task aimed at ensuring the quality of the knowledge management process by comparing the results with the expected benefits. This task needs to define and assess several metrics for that purpose.

Operational tasks of knowledge management are mainly concerned with the capture, maintenance, and use. We describe how the proposed approach support architecture knowledge capture and maintenance in section 3 and the utility of the captured and preserved knowledge is discussed in section 4.

2.4. Experience Factory Organization

The Experience Factory Organization (EFO) provides a conceptual framework for building a systematic approach to accumulate and reuse domain specific knowledge [9]. The main objective of the EFO approach to improve the performance (in terms of cost, quality, and schedule) of software development projects by leveraging experience from previous projects [8]. The EFO framework takes into account the reality that accumulating and maintaining knowledge and experiences of software development are non-trivial tasks, which should not be left to individual projects. This is because it is difficult for a project team to devote resources to capture their experiences for reuse while deadlines are looming or quality and productivity have top priority.

The EFO addresses this issue by dividing the responsibilities of software development and experience accumulation into two organizational units:

1. Project Organization: uses packaged experience to deliver software products
2. The experience Factory: supports software development by providing tailored experience [9].

Unlike the EFO, our approach treats the experience factory as a tool, called the Architecture Knowledge Repository (AKR), instead of a separate organizational unit. However, the AKR has also been logically divided into project knowledge (concrete) and corporate knowledge (generic). Another requirement of reusability is an appropriate structure to enable tailoring and generalizing knowledge. We have addressed this issue by designing a set of templates to capture and present architecture design knowledge [5].

3. Capturing Architecture Knowledge

This section presents a conceptual framework for capturing implicit knowledge. This framework provides a support mechanism to design, develop and populate a knowledge repository to improve architecture processes. The proposed framework comprises planning, capturing, organization and evaluation, and storage of architecture knowledge (Figure 3).

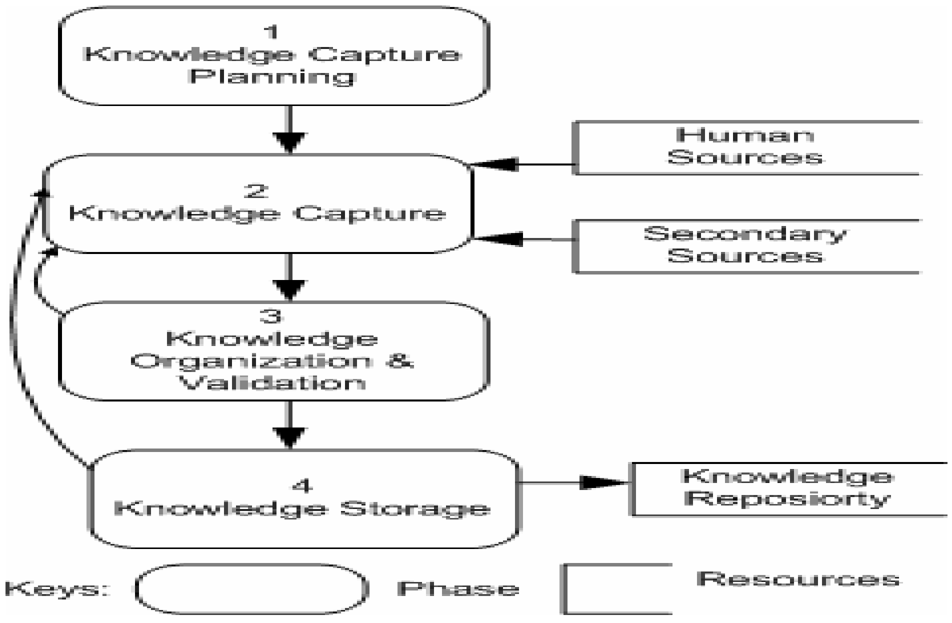


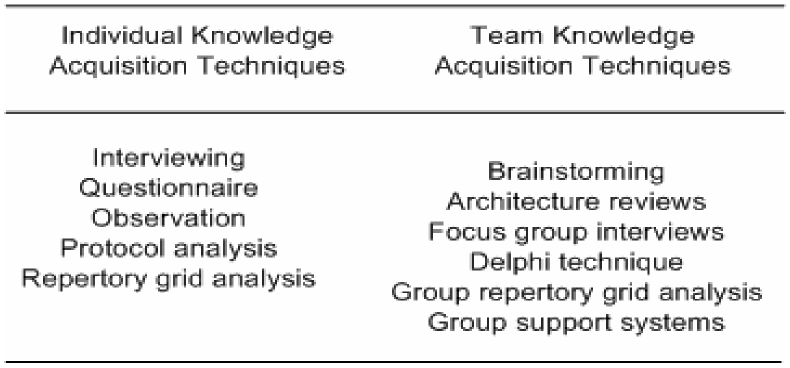
Figure 3: A conceptual framework for capturing architecture knowledge

The planning phase is aimed at understanding the knowledge domain, identifying the sources of the knowledge, and deciding about the techniques to be used. The main objective of knowledge capture phase is to acquire knowledge from human or secondary sources using the techniques described in section 3.1 and 3.2. The knowledge captured in this phase needs to be organized and evaluated (the objective of phase 3) before being placed in the AKR as a reusable artifact. There are different techniques (such as transcription, coding, summarization [30]) to organize knowledge depending on the knowledge capture source and methods. For example, we have developed different templates to organize knowledge extracted from patterns[3]. The organized knowledge is validated before being stored in knowledge repository. The cycle between phase 2-4 runs until most of the required implicit knowledge from the people or secondary sources has been extracted, organized and stored.

3.1. Capturing Knowledge from Human Sources

One of the main sources of implicit architecture knowledge is people (e.g. architects, domain experts and project teams), who individually and collectively carry a large amount of "*know-how*" and "*community specific folklore*" about their domain and projects [42]. There are two main strategies to capture such implicit knowledge to populate a knowledge repository: 1) appoint a knowledge engineer to capture implicit knowledge from individuals or teams [41, 42] or 2) provide appropriate tool support so that knowledge can be encoded into the system as part of the knowledge creation process. The latter is called *contextualized* knowledge acquisition [24]. This strategy is similar to Electronic Process Guide (EPG) [39]. It is not the intent of this paper to recommend a particular strategy as each of them have been found useful in different contexts.

Table 1: Some Knowledge Acquisition Techniques



Individual Knowledge Acquisition Techniques	Team Knowledge Acquisition Techniques
Interviewing	Brainstorming
Questionnaire	Architecture reviews
Observation	Focus group interviews
Protocol analysis	Delphi technique
Repertory grid analysis	Group repertory grid analysis
	Group support systems

When applying the first strategy of knowledge acquisition, someone can use a variety of techniques derived from different disciplines such as expert systems, artificial intelligence, groupware systems and others. Table 1 presents some of the techniques that are useful to capture implicit knowledge. A succinct explanation of these techniques is provided in [32].

To implement the second strategy, a suitable environment is provided so that knowledge generators can encode the knowledge in a system as it is created [24]. We have developed a knowledge repository as a support mechanism for this strategy. However, an empty knowledge repository cannot motivate people to use it. Before exposing the potential users to a knowledge repository, it should be populated [38]. This can be done by capturing knowledge from experts using the above-mentioned techniques or from secondary sources such as patterns. We have developed a "*pattern-mining*" approach to populate the AKR.

3.2. Capturing Knowledge from Patterns

We have found that software patterns are a valuable source of architecturally significant constructs (such as scenarios and tactics) and relationships between them. These synergistic relationships should be captured and documented as reusable architecture knowledge to support and improve architecting activities [3, 5]. To facilitate the task of knowledge acquisition from patterns, we have developed:

- a process model to capture and structure architecture knowledge from patterns.
- a set of guidelines to identify and capture the architectural information that can be captured as a reusable artifact from a pattern.
- a set of templates to structure and document the extracted architecture knowledge.

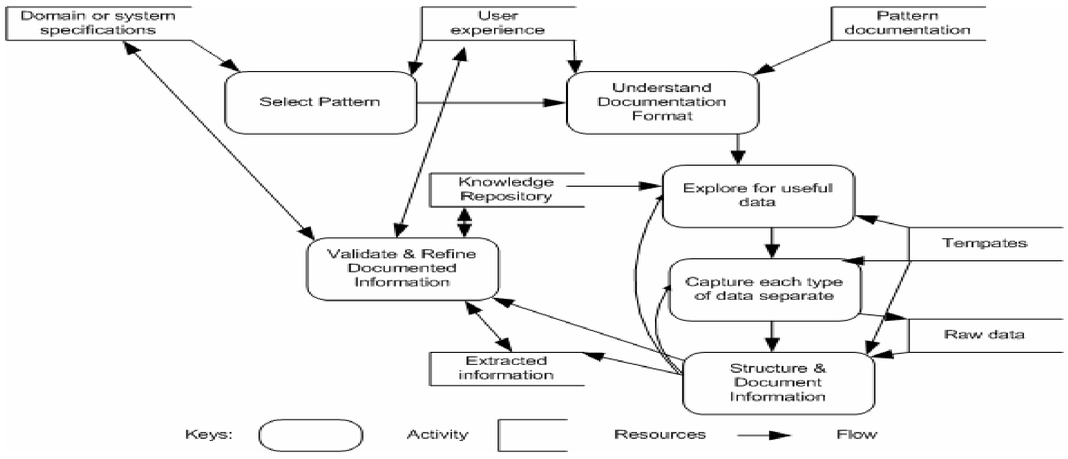

Domain or system specifications
User experience
Pattern documentation
Select Pattern
Understand Documentation Format
Knowledge Repository
Explore for useful data
Validate & Refine Documented Information
Tempates
Capture each type of data separate
Raw data
Extracted information
Structure & Document Information
Keys:
Activity
Resources
Flow

Figure 4: A process model of mining patterns for architecture knowledge from patterns

In the following, we describe the steps of the pattern-mining process (Figure 4).

The process consists of the following steps:

1. Select a software pattern to be explored for architectural information. This decision is usually influenced by a system's domain and the software engineer's experience.
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 using the provided template
6. Validate and refine documented information based on domain knowledge and experience of using different patterns.

Patterns are usually documented in a variation of the format used in [13, 19]. This requires the inclusion of problem, solution, and quality consequences parts. Figure 5 presents a diagrammatic guide to spot architecturally significant information from a pattern. Our experience is that scenarios are mostly found in the problem and solution sections. A pattern's forces can also be found in these sections. However, there may be a separate section for describing forces. The quality attributes (positively or negatively affected) are described in the quality consequence section, usually at the end of a pattern's description.

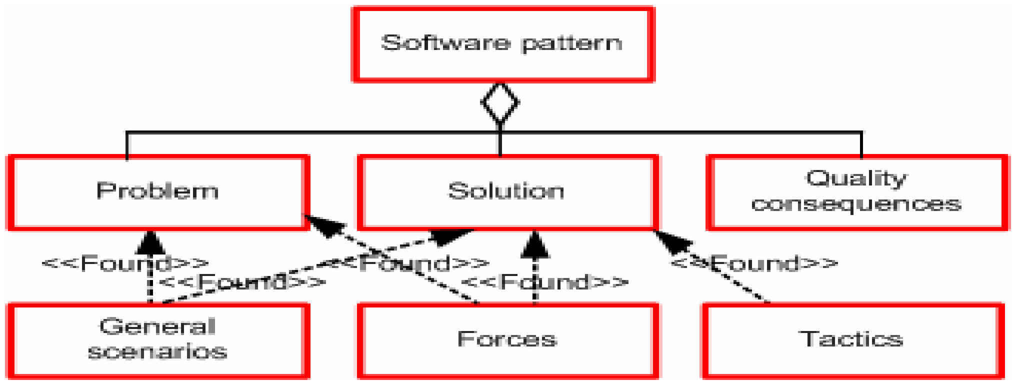


Figure 5: A simple guide to spot architecturally significant information in a pattern

The extracted information must also be structured and documented in a format that creates a readily useable knowledge artifact. We have designed a set of templates to document different units of architecturally significant information (i.e. general scenarios, quality attributes, tactics, usage examples and so on) as an artifact of architecture knowledge. Table 2 presents one of these templates. The template presents different pieces of a pattern's description in a succinct format at an abstraction level suitable for architecting activities, where abstract scenarios are used to characterize required quality attributes and suitable patterns are chosen based on their support for the required quality attributes.

Table 2: A template to document architectural knowledge extracted from patterns

Pattern Name: *Name of the software pattern*			**Pattern Type:** *Architecture, design, or style*
Brief description		*A brief description of the pattern.*	
Context		*The situation for which the pattern is recommended.*	
Problem description		*What types of problem the pattern is supposed to address?*	
Suggested solution		*What is the solution suggested by the pattern to address the problem?*	
Forces		*Factors affecting the problem and solution. Justification for using pattern.*	
Available tactics		*What tactics are used by the pattern to implement the solution?*	
Affected Attributes		**Positively**	**Negatively**
		Attributes supported	*Attributes hindered*
General scenarios	S1	*A textual, system independent specification of a quality attribute.*	
	S..n		
Usage examples	*Some known examples of the usage of the pattern to solve the problems.*		

The template presented in Table 2 makes the relationships among scenarios, quality attributes, and patterns explicit. Moreover, it also captures one of the most important parts of a pattern description, namely the *forces*. The forces of a pattern are usually described implicitly in most of the pattern documentation styles. Recently, there are some efforts to pay more attention to the forces of a pattern [21, 26].

Table 3: A template to document architecture knowledge for SA evavluation process

Project Name: *Which project needs this scenario?* **Project domain:** *Domain of the project*		**Date:** *When was proposed?* **Scenarios No:** *Serial number assigned to the scenario*
Business goals		*Which business goals does this scenario achieve?*
Stakeholders		*Which class of the stakeholders did suggest this scenario?*
Attributes		*Which quality attributes are required by this scenario?*
Description		*A brief description of the scenario.*
Concrete scenario	**Stimulus**	*A condition that needs to be considered when it arrives at a system.*
	Context	*A system's condition when a stimulus occurs, e.g. overloaded, running etc.*
	Response	*A measurable action that needs to be undertaken after the arrival of the stimulus*
	Complexity	*How complex is this scenario to realize? (Effect on macro or micro architecture)*
	Priority	*How important is this scenario?*
Pattern/Style		*Name of the architectural pattern or style that can support this scenario.*
Design tactics		*What are the design tactics used by the pattern/style to support the scenarios?*
Design rational		*What are reasons for using the patterns/tactics? How does it provide the desired quality attributes?*

The abstract knowledge captured with the template 1 can be concretized for a specific project. For example, general scenarios are concretized to specify quality attributes. Table 3 presents the second template for documenting architecture design knowledge for supporting architecture evaluation, which needs concrete scenarios along with other information (e.g. level of complexity and importance). Scenario-based approaches mainly gather scenarios from stakeholders. We have found that many concrete scenarios can be derived from the abstract scenarios extracted from patterns. It also increases confidence in an architecture's capability of satisfying certain concrete scenarios if these scenarios are instances of the general scenarios extracted from a pattern used in that architecture [5].

Table 4: Abstract architecture knowledge extracted from J2EE Business Delegate pattern

Pattern Name: *Business Delegate*			**Pattern Type:** *Design pattern*
Brief description		*This pattern reduces coupling between tiers by providing an entry point for accessing the services another tier. It also supports results caching to improve performance…*	
Context		*A client may be exposed to the complexity of dealing with the distributed components…*	
Problem description		*Presentation-tier components interact directly with business services. Such a direct interaction makes the clients vulnerable to any changes in the business services…*	
Suggested solution		*Reduce coupling between presentation-tier clients and business services. The Business Delegate hides the underlying implementation details of the business service...*	
Forces		*Presentation-tier clients require access to business service.* *It is desirable to minimize coupling to hide implementation details from clients.*	
Available tactics		*Delegate Proxy and Delegate Adapter*	
Affected Attributes		**Positively**	**Negatively**
		Reduce coupling, manageability, performance	*Introduce new layer, increased complexity*
General scenarios	S1	*Presentation-tier components shall not be exposed to the implementation details of the business services they use.*	
	S2	*System shall provide a caching mechanism to improve response to business service request.*	
	S3	*Services calls across network or tiers shall be minimized to avoid degraded performance.*	
Examples	*E-commerce portals, online content providers, sports websites.*		

Table 4 demonstrates how the template (Table 2) can be used to structure and maintain abstract architecture knowledge extracted from patterns. Table 4 contains the knowledge extracted from Business Delegate J2EE pattern by following the pattern-mining process.

Table 5 demonstrates how the abstract knowledge captured by Table 4 can be concretized to evaluate an architecture that is using Business Delegate pattern.

Table 5: A template to document architecture knowledge for software architecture evavluation

Project Name: *Qualification Verification System* **Project domain:** *E-Commerce application*		**Date:** *12/06/2005* **Scenarios No:** *Serial number assigned to the scenario*
Business goals	*Customer satisfaction and process efficiency.*	
Stakeholders	*Business Manager, System sponsors, and End User.*	
Attributes	*Improved performance*	
Description	*The response to a business service request shall be improved to avoid users' frustration and system shall be able to handle up to 1000 users concurrently without any delay in the response time.*	
Concrete scenario	**Stimulus**	*A user request needs to be processed.*
	Context	*There are 1000 users, who may make simultaneous requests.*
	Response	*The system shall be able to respond to a request within seconds.*
	Complexity	*Medium*
	Priority	*High*
Pattern/Style	*Business Delete*	
Design tactics	*Delegate proxy and Caching*	
Design rational	*This pattern exposes an interface to the business service API by using proxy function to pass the client methods to the session bean. It can cache any necessary data and references to the session bean's home or remote objects to improve performance by reducing the number of lookups.*	

4. Utility of Architecture Knowledge

In this section, we discuss the potential utility of the architecture knowledge captured from human sources or patterns, structured using the proposed templates, and stored and managed by an architecture knowledge repository. We believe that the availability of such a knowledge repository can improve architecture-based software development process by providing a support mechanism to capture, store, and retrieve the required architecture knowledge. For example, a design team can benefit from such a tool by logging unsolved issues to be discussed and resolved during subsequent design meetings. We have reproduced the Figure 1, architecture-based software development process model, in Figure 6 to demonstrate the support provided by the AKR to different activities of the process.

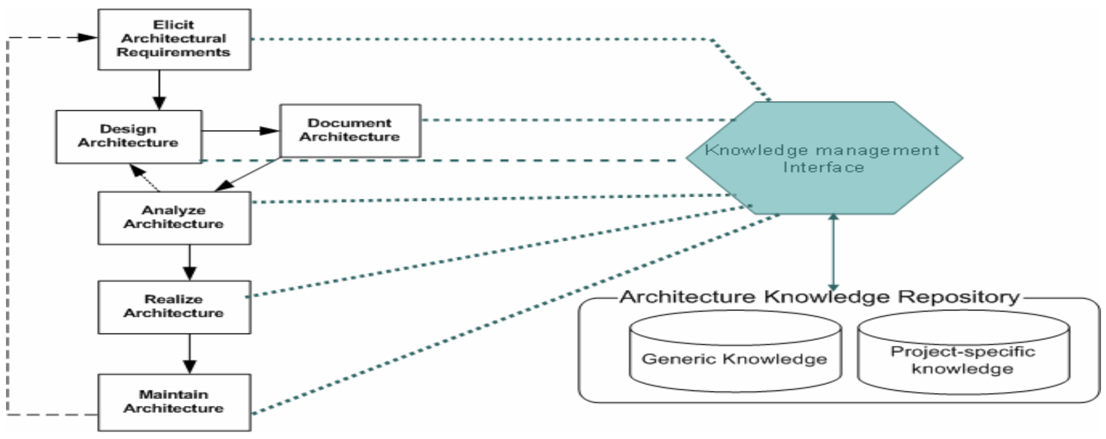

Elicit Architectural Requirements
Design Architecture
Document Architecture
Analyze Architecture
Realize Architecture
Maintain Architecture
Knowledge management Interface
Architecture Knowledge Repository
Generic Knowledge
Project-specific knowledge

Fig. 6: Supporting architecture-based development with the design experience repository

Architecture is usually designed iteratively by devising and reasoning about design decisions with respect to the quality sensitive scenarios created during the architecture requirements elicitation stage. Design decisions usually apply several architecture or design patterns to achieve the desired set of quality attributes [11]. Knowledge of different technical and functional domains is considered the raw material for architecture design process. That is why a project manager attempts to staff a design team with people who can match the knowledge needs of a project [45]. However, this seldom happens because of several reasons such as shortfall of technical knowledge, thin distribution of domain knowledge, staff attrition or movement and others. Thus, a design team's main responsibilities includes knowledge acquisition, knowledge sharing and knowledge integration [45], which can be supported by the architecture knowledge repository populated by architecture knowledge captured by following the techniques discussed in section 3.

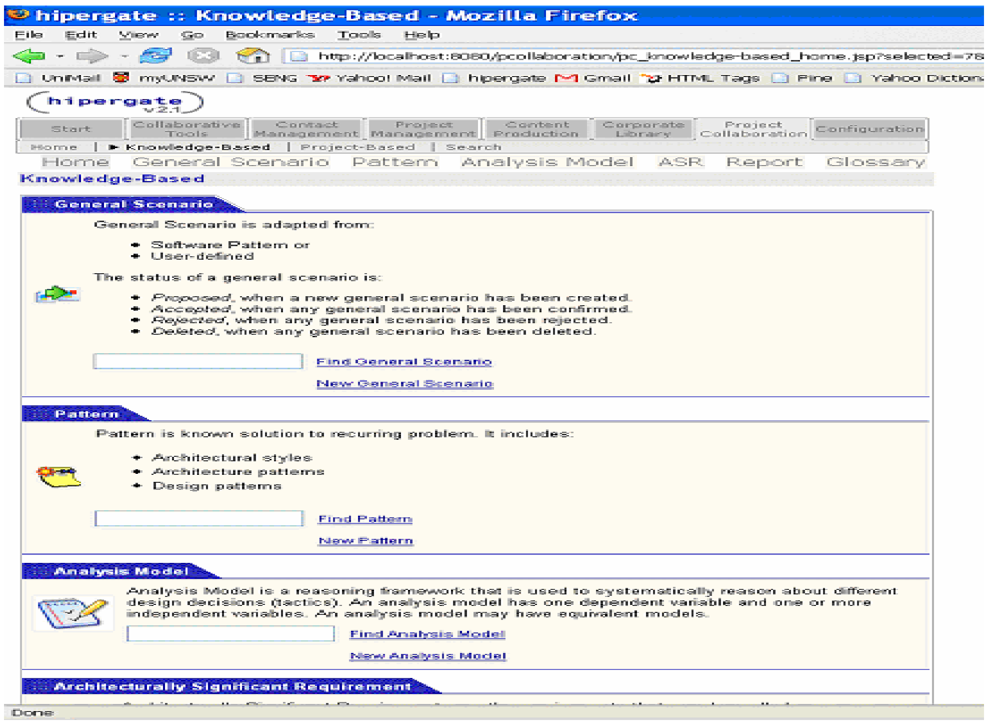
Architects describe architectural decisions using different views such as procedural, concurrency, code, development and others [11]. However, the knowledge about the process that leads to a particular design decision is usually not captured [12], which results in several problems discussed in Section 2.2. The AKR provides an environment to capture and manage not only architecture design decision but also process knowledge to support subsequent activities of architecture-based development. On the reusability of the design knowledge, the generic architecture knowledge along with the contextual information should help architects identify suitable patterns by comparing the scenarios and quality attributes supported by different patterns with the ones required by the stakeholders. Moreover, architects can also evaluate the suitability of generic architecture decisions suggested for a particular context and they can contact the contributor of a particular architecture decision for further explanation.

Software architecture evaluation activities can also be improved by using both generic and project-specific knowledge about architecture artifacts and processes leading to those artifacts. For example, generic architecture knowledge can help improve the task of specifying quality attributes using scenario, select suitable reasoning frameworks to be used to assess certain design decisions with respect to the desired quality attributes and increase confidence in the capabilities of architecture to satisfy particular quality sensitive scenarios as a result of using certain patterns [3, 47].

Project-specific knowledge helps designers, developers and maintainers to better understand the architecture decisions, their constraints and reasoning behind it. Moreover, the availability of the reasoning behind the architectural decisions helps architects explain architectural choices and how they satisfy business goals [44]. Such knowledge is also valuable during the architecture realization and maintenance stages (Figure 6) of architecture-based development processes. For example, if the rationales underpinning different design decisions are available, developers can gain invaluable insights into the potential implications of different implementation choices for architectural decisions. Moreover, architects themselves also need the architecture design process knowledge in order to avoid the path they would have considered and discarded. We have designed an empirical research program to assess different uses of the knowledge captured from pattern and preliminary results are very encouraging [5].

5. PAKME – Process-centric Architecture Knowledge Management Environment

In this section, we briefly introduce a prototype tool that we have been developing to demonstrate the feasibility of the proposed conceptual framework for capturing and managing architecture design knowledge. The Process-based Architecture Knowledge Management Environment (PAKME) is a prototype web-based system to provide knowledge management support for improving architecture-based software development process. The PAKME has been built on top of an open source groupware platform, Hipergate [1]. This provides various collaborative features including contact management, project management, online collaboration tools and others. We have modified the data model of the Hipergate to add the features required to capture, manage, and retrieve architecture knowledge captured from human sources and patterns. The AKR database consists of 25 tables to store different types of architectural artifacts and rationales.


hipergate :: Knowledge-Based - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/pcollaboration/pc_knowledge-based_home.jsp?selected=78
UniMail myUNSW SENG Yahoo! Mail hipergate Gmail HTML Tags Pine Yahoo Diction
hipergate v2.1
Start Collaborative Tools Contact Management Project Management Content Production Corporate Library Project Collaboration Configuration
Home | Knowledge-Based | Project-Based | Search
Home General Scenario Pattern Analysis Model ASR Report Glossary
Knowledge-Based
General Scenario
Find General Scenario
New General Scenario
Pattern
Find Pattern
New Pattern
Analysis Model
Find Analysis Model
New Analysis Model
Architecturally Significant Requirement
Done

General Scenario is adapted from:

- Software Pattern or
- User-defined

The status of a general scenario is:

- *Proposed*, when a new general scenario has been created.
- *Accepted*, when any general scenario has been confirmed.
- *Rejected*, when any general scenario has been rejected.
- *Deleted*, when any general scenario has been deleted.

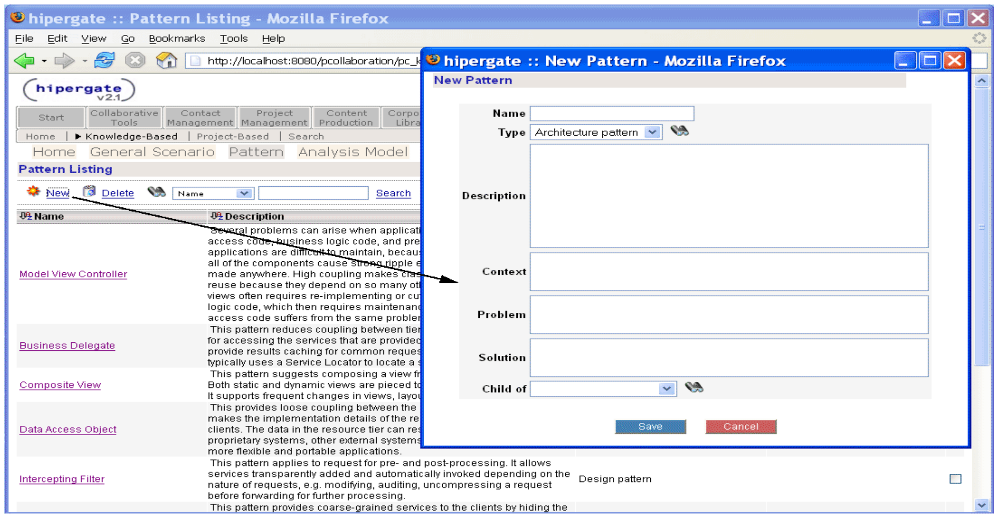
Pattern is known solution to recurring problem. It includes:

- Architectural styles
- Architecture patterns
- Design patterns

Analysis Model is a reasoning framework that is used to systematically reason about different design decisions (tactics). An analysis model has one dependent variable and one or more independent variables. An analysis model may have equivalent models.

Figure 7: Front page of knowledge-based repository

The knowledge repository is logically divided into *knowledge-based artifacts*, *generic knowledge, and project-based artifacts.* The generic knowledge is accumulated by using the implicit knowledge capture techniques described in this paper. So far we have populated the ARK by distilling architecture knowledge from several J2EE [6] patterns, architecture patterns [13], and the Battle Control System (BCS) case study described in [15]. Project-based architecture knowledge consists of the artifacts either instantiated from generic knowledge or newly created during various architecture activities. Figure 7 shows the front page of knowledge-based of the AKR.

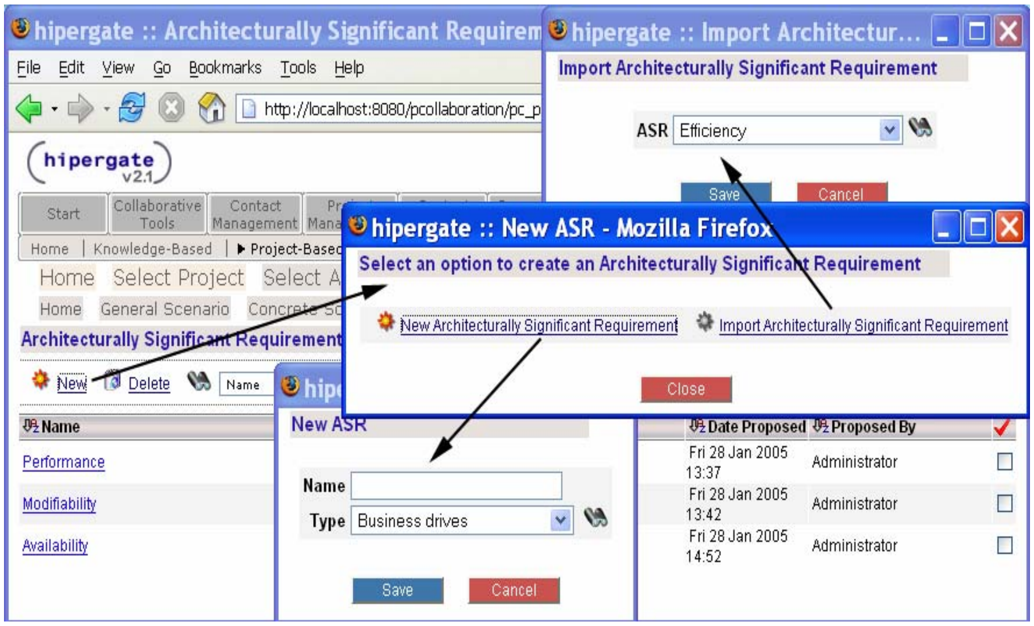


Type: Architecture pattern

Name	Description	
Model View Controller	Several problems can arise when applicati access code, business logic code, and pre applications are difficult to maintain, becau all of the components cause strong ripple e made anywhere. High coupling makes clas reuse because they depend on so many ot views often requires re-implementing or cu logic code, which then requires maintenanc access code suffers from the same probler	
Business Delegate	This pattern reduces coupling between tier for accessing the services that are providec provide results caching for common reques typically uses a Service Locator to locate a s	
Composite View	This pattern suggests composing a view fr Both static and dynamic views are pieced tc It supports frequent changes in views, layou	
Data Access Object	This provides loose coupling between the makes the implementation details of the re clients. The data in the resource tier can res proprietary systems, other external systems more flexible and portable applications.	
Intercepting Filter	This pattern applies to request for pre- and post-processing. It allows services transparently added and automatically invoked depending on the nature of requests, e.g. modifying, auditing, uncompressing a request before forwarding for further processing.	Design pattern
	This pattern provides coarse-grained services to the clients by hiding the	

Figure 8: A form for entering a new pattern in the AKR

Currently, the PAKME consists of four components; knowledge acquisition, knowledge maintenance, knowledge retrieval, and knowledge presentation. The knowledge acquisition component provides various forms and editing tools to enter new generic or project-specific knowledge in the repository. The forms are based on the templates (e.g. Table 2) developed to organize knowledge. Figure 8 shows a form for entering a new pattern in the AKR. While entering a new artifact, an end user can view the existing artifacts in the background as shown in Figure 8. For example, if a user's search fails to retrieve a particular pattern, the user may decide to enter that pattern in the repository.



hipergate :: Architecturally Significant Requirem
File Edit View Go Bookmarks Tools Help
http://localhost:8080/pcollaboration/pc_p
hipergate v2.1
Start
Collaborative Tools
Contact Management
Home
Knowledge-Based
Project-Based
Home
Select Project
Home
General Scenario
Architecturally Significant Requirement
New
Delete
Name
hipergate :: Import Architectur...
Import Architecturally Significant Requirement
ASR
Save
Cancel
hipergate :: New ASR - Mozilla Firefox
Select an option to create an Architecturally Significant Requirement
New Architecturally Significant Requirement
Import Architecturally Significant Requirement
Close
New ASR
Name
Type
Save
Cancel

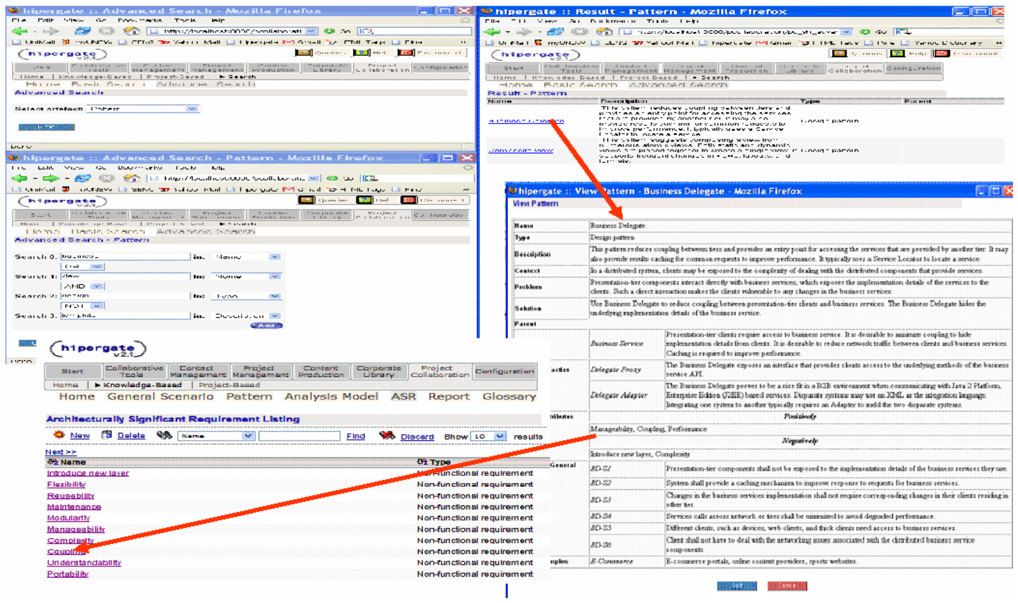
ASR: Efficiency

Type: Business drives

Name	Date Proposed	Proposed By
Performance	Fri 28 Jan 2005 13:37	Administrator
Modifiability	Fri 28 Jan 2005 13:42	Administrator
Availability	Fri 28 Jan 2005 14:52	Administrator

Figure 9: Screen shots showing the search and navigation based retrieval from the AKR.

The knowledge acquisition component for project-specific knowledge provides various features to acquire new architectural artifacts (such as scenarios, architecturally significant requirements, design decisions and others) or import the generic artifacts for a particular project. Figure 9 shows that a user can either enter a new ASR or import an existing ASR in a project. The maintenance component provides various features to modify, delete and instantiate different artifacts. It also includes repository administration functions.


hipergate :: Advanced Search - Mozilla Firefox
hipergate :: Result - Pattern - Mozilla Firefox
hipergate :: Advanced Search - Pattern - Mozilla Firefox
hipergate :: View Pattern - Business Delegate - Mozilla Firefox
hipergate v2.1
Start
Collaborative Tools
Contact Management
Project Management
Content Production
Corporate Library
Project Collaboration
Configuration
Home | Knowledge-Based | Project-Based
Home General Scenario Pattern Analysis Model ASR Report Glossary
Advanced Search
Advanced Search - Pattern
Result - Pattern
Add
New Delete Find Discard Show 10 results
Next >>
View Pattern

Architecturally Significant Requirement Listing

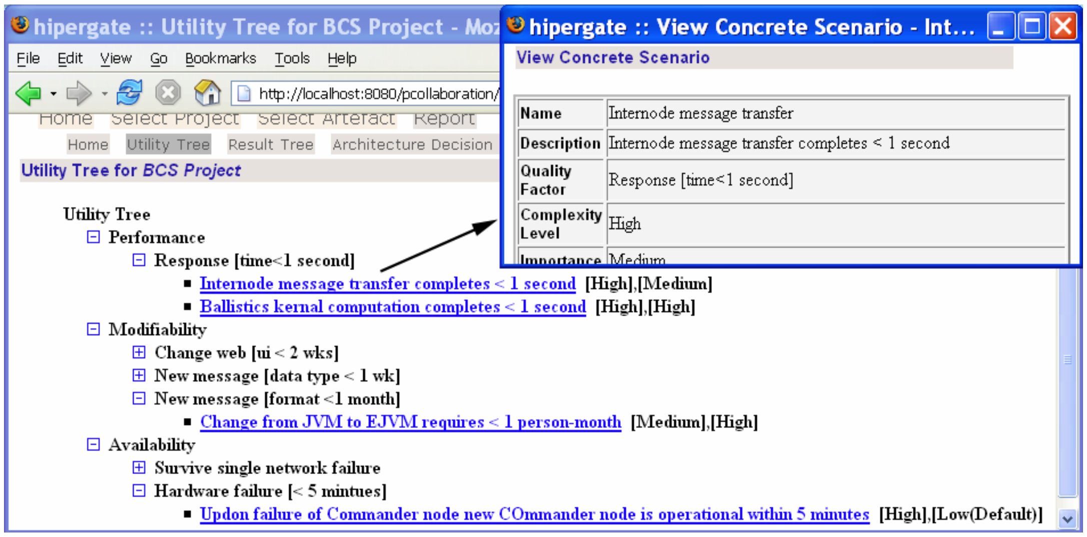
Name	Type
Introduce new layer	Non-functional requirement
Flexibility	Non-functional requirement
Reusability	Non-functional requirement
Maintenance	Non-functional requirement
Modularity	Non-functional requirement
Manageability	Non-functional requirement
Complexity	Non-functional requirement
Coupling	Non-functional requirement
Understandability	Non-functional requirement
Portability	Non-functional requirement

Name	Business Delegate	
Type	Design pattern	
Description	This pattern reduces coupling between tiers and provides an entry point for accessing the services that are provided by another tier. It may also provide results caching for common requests to improve performance. It typically uses a Service Locator to locate a service.	
Context	In a distributed system, clients may be exposed to the complexity of dealing with the distributed components that provide services.	
Problem	Presentation-tier components interact directly with business services, which exposes the implementation details of the services to the clients. Such a direct interaction makes the clients vulnerable to any changes in the business services.	
Solution	Use Business Delegate to reduce coupling between presentation-tier clients and business services. The Business Delegate hides the underlying implementation details of the business service.	
Parent		
	Business Service	Presentation-tier clients require access to business service. It is desirable to minimize coupling to hide implementation details from clients. It is desirable to reduce network traffic between clients and business services. Caching is required to improve performance.
actics	Delegate Proxy	The Business Delegate exposes an interface that provides clients access to the underlying methods of the business service API.
	Delegate Adapter	The Business Delegate proves to be a nice fit in a B2B environment when communicating with Java 2 Platform, Enterprise Edition (J2EE) based services. Disparate systems may use an XML as the integration language. Integrating one system to another typically requires an Adapter to meld the two disparate systems.
tributes	*Positively*	
	Manageability, Coupling, Performance	
	Negatively	
	Introduce new layer, Complexity	
General	BD-S1	Presentation-tier components shall not be exposed to the implementation details of the business services they use.
	BD-S2	System shall provide a caching mechanism to improve response to requests for business services.
	BD-S3	Changes in the business services implementation shall not require corresponding changes in their clients residing in other tier.
	BD-S4	Services calls across network or tiers shall be minimized to avoid degraded performance.
	BD-S5	Different clients, such as devices, web clients, and thick clients need access to business services.
	BD-S6	Client shall not have to deal with the networking issues associated with the distributed business service components.
mples	E-Commerce	E-commerce portals, online content providers, sports websites.

Figure 10: Screen shots showing the search and navigation based retrieval from the AKR.

The retrieval component supports both basic and advanced searches to find and retrieve the desired architecture artifacts and rationales. For example, a user can perform a search for a suitable pattern to satisfy a particular quality attribute or to find a design decisions suggested for a particular domain/context by a certain designer. To facilitate the search based on keywords, the AKR allows the users to associated different keywords to each architecture artifact when that artifact is entered in the repository or later on. The retrieval component also enables a user to traverse to different related artifacts by navigating through the knowledge space based on the initial results of a search query (Figure 10). Advanced search facility enables a users to use logical operators (such as And, Or, Not) to include or exclude certain architecture artifacts in the search results.

The knowledge presentation component supports generating different views of the architecture knowledge residing in the AKR. For example, it presents utility (Figure 11) tree to specify quality attributes along with their respective priority and level of complexity and result tree based on the results of architecture evaluation sessions using a scenario based evaluation method like ATAM [11] or SAAM [15].


hipergate :: Utility Tree for BCS Project - Moz
File Edit View Go Bookmarks Tools Help
http://localhost:8080/pcollaboration/
Home Select Project Select Artefact Report
Home Utility Tree Result Tree Architecture Decision
hipergate :: View Concrete Scenario - Int...
View Concrete Scenario

Utility Tree for *BCS Project*

Utility Tree
- Performance
  - Response [time<1 second]
    - Internode message transfer completes < 1 second [High],[Medium]
    - Ballistics kernal computation completes < 1 second [High],[High]
- Modifiability
  - Change web [ui < 2 wks]
  - New message [data type < 1 wk]
  - New message [format <1 month]
    - Change from JVM to EJVM requires < 1 person-month [Medium],[High]
- Availability
  - Survive single network failure
  - Hardware failure [< 5 mintues]
    - Updon failure of Commander node new COmmander node is operational within 5 minutes [High],[Low(Default)]

Name	Internode message transfer
Description	Internode message transfer completes < 1 second
Quality Factor	Response [time<1 second]
Complexity Level	High
Importance	Medium

Figure 11: A utility tree of concrete scenarios and their priorities and complexity.

To summarize, the two main objectives of the PAKME are:

- To provide a support mechanism for capturing, managing, and retrieving architecture knowledge to improve the quality of architecture activities.
- To act as a source of architecture knowledge for those who need rapid access to experience-based design decisions to assist in making new decisions or discovering the rationale for past decisions.

6. Conclusion and Future Work

Our research is aimed at improving the effectiveness of SA processes by providing suitable support mechanisms. Current approaches are deficient in providing the required design knowledge or managing the knowledge generated. This leads to a lack of use of existing SA knowledge as it is not available in a readily usable format at an appropriate level of abstraction. Moreover, implicit knowledge is not normally captured to make it available for decision support.

This paper emphasizes the importance of capturing and using implicit software architecture design knowledge to improve architecture activities. We present a framework for capturing implicit knowledge using various knowledge acquisition and pattern-mining techniques and structuring and storing that knowledge in a knowledge repository developed to support the proposed framework. This framework supports the strategic and operational tasks of architecture knowledge management model presented in Section 2.3.

Future work includes enhancement of the tool with case-based approaches [24] and incremental refinement of search queries based on the results of the basic search. We are particularly keen to test the pattern-mining process and tool in industrial settings, so that their applicability and scalability can be thoroughly assessed. The preliminary results of our assessment of the pattern-mining process and the usefulness of the extracted knowledge are very encouraging[5]. These give us confidence in the utility of our approach.

Acknowledgement: *Several undergraduate students helped us build the tool. National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.*

References

[1] Hipergate - Open Source CRM and Groupware. Last accessed on 16th April, 2005, Available from: http://www.hipergate.com.
[2] Al-Naeem, T., et al. A Quality-Driven Systematic Approach for Architecting Distributed Software Applications. 27th Int'l. Conf. on Software Eng. 2005. St. Louis, USA.
[3] Ali-Babar, M. Scenarios, Quality Attributes, and Patterns: Capturing and Using their Synergistic Relationships for Product Line Architectures. Int,l. Workshop on Adopting Product Line Software Engineering. 2004. Busan, South Korea.
[4] Ali-Babar, M., I. Gorton, and B. Kitchenham, A Framework for Supporting Architecture Knowledge and Rationale Management, in Rationale Management in Software Engineering, A.H. Dutoit, et al., Editors. 2005, Submitted for review.
[5] Ali-Babar, M., et al. Mining Patterns for Improving Architecting Activities - A Research Program and Preliminary Assessment. 9th Int'l. conf. on Empirical Assessment in Software Engineering. 2005. Keele, UK.
[6] Alur, D., J. Crupi, and D. Malks, Core J2EE Patterns: Best Practices and Design Strategies. 2nd ed. 2003: Sun Microsystem Press.
[7] Bachmann, F., L. Bass, and M. Klein, Deriving Architectural Tactics: A Step toward Methodical Architectural Design, Tech Report CMU/SEI-2003-TR-004, SEI, Carnegie Mellon University, USA, 2003
[8] Basili, V.R. and G. Caldiera, Improving Software Quality Reusing Knowledge and Experience. Sloan Management Review, 1995. **37**(1): p. 55-64.
[9] Basili, V.R., G. Caldiera, and H.D. Rombach, The Experience Factory, in Encyclopedia of Software Engineering, J.J. Marciniak, Editor. 2001, John Wiley & Sons.
[10] Bass, L. and R. Kazman, Architecture-Based Development, Tech Report CMU/SEI-99-TR-007, SEI, Carnegie Mellon University, Pittsburgh, 1999
[11] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice. 2 ed. 2003: Addison-Wesley.
[12] Bosch, J. Software Architecture: The Next Step. European Workshop on Software Architecture. 2004.
[13] Buschmann, F., et al., Pattern-Oriented Software Architecture: A System of Patterns. 1996: John Wiley & Sons.
[14] Clements, P., et al., Documenting Software Architectures: Views and Beyond. 2002: Addison-Wesley.
[15] Clements, P., R. Kazman, and M. Klein, Evaluating Software Architectures: Methods and Case Studies. 2002: Addison-Wesley.
[16] Davenport, T.H. and L. Prusak, Working Knowledge. 1998: Harvard Business School Press, Boston, Massachusetts.
[17] Desouza, K.C. and J.R. Evaristo, Managing Knowledge in Distributed Projects. Communication of the ACM, 2004. **47**(4): p. 87-91.
[18] Dutoit, A.H. and B. Paech, Rationale Management in Software Engineering, in Handbook of Software Engineering and Knowledge Engineering, S. Change, Editor. 2001, World Scientific Publishing, Singapore.
[19] Gamma, E., et al., Design Patterns-Elements of Reusable Object-Oriented Software. 1995, Reading, MA: Addison-Wesley.
[20] Gorton, I. and J. Haack. Architecting in the Face of Uncertainty: An Experience Report. Proc. International Conference on Software Engineering. 2004. Edinburgh, Scotland.
[21] Gross, D. and E. Yu. From Non-Functional Requirements to Design through Patterns. 6th Int'l Workshop on Requirements Engineering Foundation for Software Quality. 2000. Sweden.
[22] Gruber, T.R. and D.M. Russell, Design Knowledge and Design Rationale: A Framework for Representing, Capture, and Use, Tech Report KSL 90-45, Knowledge Systems Laboratory, Standford University, California, USA, 1991
[23] Hansen, M.T., N. Nohria, and T. Tierney, What's your strategy for managing knowledge? Harvard Business Review, March-April 1999: p. 106-116.
[24] Henninger, S., Tool Support for Experience-Based Software Development Methologies. Advances in Computers, 2003. **59**: p. 29-82.

[25] Jarczyk, A.P.J., P. Loffler, and F.M.S. III. Design Rationale for Software Engineering: A Survey. Proc. 25th Hawaii Int'l. Conf. on System Sciences. 1992.
[26] John, B.E., et al. Bringing Usability Concerns to the Design of Software Architecture. 9th IFIP Working Conference on Engineering for Human-Computer Interaction. 2004. Hamburg, Germany.
[27] Kazman, R., et al., Scenario-Based Analysis of Software Architecture. IEEE Software Engineering, Nov. 1996.
[28] Kruchten, P., The Rational Unified Process: An Introduction. 2nd ed. 2000: Addison-Wesley.
[29] Kruchten, P.B., The 4+1 View Model of architecture. Software, IEEE, 1995. **12**(6): p. 42-50.
[30] Land, L.P.W., A. Aurum, and M. Handzic. Capturing Implicit Software Engineering Knowledge. Proc. 13th Australian Software Engineering Conference. 2001. Canberra, Australia.
[31] Lassing, N., D. Rijsenbrij, and H.v. Vliet, How Well can we Predict Changes at Architecture Design Time? Journal of Systems and Software, 2003. **65**(2): p. 141-153.
[32] Liou, Y.I., Collaborative Knowledge Acquisition. Expert Systems With Applications, 1992. **5**(1-2): p. 1-13.
[33] Niemela, E., J. Kalaoja, and P. Lago, Toward an Architectural Knowledge Base for Wireless Service Engineering. IEEE Transactions of Software Engineering, 2005. **31**(5): p. 361-379.
[34] Nonaka, I. and H. Takeuchi, The Knowledge-Creating Company. 1995: Oxford University Press.
[35] Probst, G.J.B. Practical Knowledge Management: A Model That Works. Last accessed on 14th March, 2005, Available from: http://know.unige.ch/publications/Prismartikel.PDF.
[36] Robillard, P.N., The role of knolwedge in software development. Communications of the ACM, 1991. **42**(1): p. 87-92.
[37] Rus, I. and M. Lindvall, Knowledge Management in Software Engineering. IEEE Software, 2002. **19**(3): p. 26-38.
[38] Schneider, K. and T. Schwinn, Maturing Experience Base Concepts at Daimler Chrysler. Software Process Improvement and Practice, 2001. **6**(2): p. 85-96.
[39] Scott, L., et al., Understanding the use of an electronic process guide. Journal of Information and Software Technology, 2002. **44**(10): p. 601-616.
[40] Shaw, M. and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. 1996, Upper Saddle River, NJ: Prentice Hall.
[41] Skuce, B., Knowledge management in software design: a tool and a trial. Software Engineering Journal, Sept. 995: p. 183-193.
[42] Terveen, L.G., P.G. Selfridge, and M.D. Long, Living Design Memory: Framework, Implementation, Lessons Learned. Human-Computer Interaction, 1995. **10**(1): p. 1-37.
[43] Tiwana, A., The Knowledge Management Toolkit: Orchestrating IT, Strategy, and Knowledge Platforms. 2nd ed. 2002: Prentice-Hall.
[44] Tyree, J. and A. Akerman, Architecture Decisions: Demystifying Architecture. IEEE Software, 2005. **22**(2): p. 19-27.
[45] Walz, D.B., J.J. Elam, and B. Curtis, Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration. Communication of the ACM, 1993. **36**(10): p. 63-77.
[46] Williams, L.G. and C.U. Smith. PASA: An Architectural Approach to Fixing Software Performance Problems. Proc. of Int'l. Conference of the Computer Measurement Group. 2002. Reno, USA.
[47] Zhu, L., M. Ali-Babar, and R. Jeffery. Mining Patterns to Support Software Architecture Evaluation. 4th Working IEEE/IFIP Conference on Software Architecture. 2004.