

Development of an Intelligent System for Architecture Design and Analysis

Jingqiu Shao, Behrouz H. Far
Department of Electrical and Computer Engineering
University of Calgary
2500, University Drive, N.W.
Calgary, Alberta, Canada, T2N 1N4
{shao, far}@enel.ucalgary.ca

Abstract

Software architecture plays a pivotal role in allowing an organization to meet its business goals, in terms of the early insights it provides into the system, the communication it enables among stakeholders, and the value it provides as a re-usable asset. Unfortunately, designing and analyzing architecture for a certain system is recognized as a hard task for most software engineers, because the process of collecting, maintaining, and validating architectural information is complex, knowledge-intensive, iterative, and error-prone. The needs of software architectural design and analysis have led to a desire to create tools to support the process. This paper introduces an intelligent system, which serves the following purposes: to obtain the meaningful non-functional requirements from users; to aid in exploring architectural alternatives, to facilitate architectural analysis.

Keywords: *Software architecture, Quality attributes, Attribute Driven Design (ADD)*

1. INTRODUCTION

Software architecture is important as a discipline because software systems are becoming too complicated to be completely designed and understood by an individual. Software architecture serves as a communication vehicle among stakeholders; Software architecture manifests the earliest design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development; Software architecture is a transferable abstraction of a system [1].

The needs of software architectural design and analysis have led to a desire to create tools to support the processes. This paper will introduce such a tool, which can obtain the meaningful non-functional requirements from users, aid in exploring architectural alternatives, and facilitate architectural analysis. Section 2 describes

the relationship between software architecture and quality attributes. Attribute Driven Design method will be introduced in the section 3. Then the overall system will be analyzed and designed in the section 4.

2. Software Architecture and Quality Attributes

2.1 The Relationship between Software Architecture and Quality Attributes

In addition to satisfying the functional requirements of a system, software developers have to address its quality attributes such as performance, modifiability, availability, and usability. The ability of a system to meet these quality attributes is largely determined by its architecture; therefore, it is very important to understand the relationship between software architecture and quality attributes [2].

Systems are frequently redesigned not because they are functionally deficient, but because they are difficult to maintain, port, or are too slow. As such, the quality attributes should be addressed as early as possible in the software lifecycle and properly built into software architecture before a detailed design proceeds on an otherwise undesirable path [3].

2.2 Quality attribute scenarios

How to express the quality attributes in terms of meaningful architectural strategies or components? What does it mean to say that a system is modifiable, reliable, maintainable or secure? We use the concept of a general scenario to describe what achieving a quality attribute goal means [6]. A general scenario is a precise system-independent specification of a type of quality attribute requirement that consists of six parts: source, stimulus, artifact, and environment, response and response measure [2].

Table 1 Quality attribute scenarios

	Source	Stimulus	Artifact	Environment	Response	Response measure
General scenario	outside the system	stochastic events arrive	system	normal operation	execute its response to the event	within a specified time interval
Non-functional requirement	more than 10000 users	visit the website	web server	normal operation	requests are processed	with average latency of one second

The general scenario provides a template for a class of non-functional requirements, and these specific non-functional requirements should be instances of the general scenarios. Table 1 shows one example of Performance general scenario and an instance of this Performance general scenario.

3. Attribute Driven Design Method

To remedy the problems inherent in ad hoc architectural design, a more disciplined approach is needed to improving our ability to understand the high level system constraints and the rationale behind architectural choices, to reuse architectural design knowledge, to make the system more evolvable, to analyze the design with respect to design criteria [3].

The Attribute Driven Design (ADD) method is an approach to define the software architecture by basing the design process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage in the decomposition, tactics and architecture patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the component and connector types provided by the tactics or patterns [5].

ADD is a method for designing architectures that is based on isolating the “driving” quality requirements, identifying design decisions to satisfy them, and using those decisions as a constraint for further design decisions. The ADD method can be viewed as an extension to most other development methods, such as the Rational Unified Process. This is one of the reasons why we choose this method, since most system design and analysis are followed by Rational Unified Process. The Rational Unified Process has several steps that result in the high-level design of an architecture but then proceeds to detailed design and implementation. Incorporating ADD into it involves modifying the steps dealing with the high-level design of the architecture and then following the process as described by Rational.

Steps of ADD [5]:

1. Choose the design element to decompose. The design element to start with is usually the whole system.

2. Refine the element according to these steps:

A. Choose the architectural drivers from the set of quality scenarios and functional requirements.

B. Choose an architectural pattern or style that satisfies the architectural drivers.

C. Instantiate children design elements and allocate functionality from use cases.

3. Repeat the steps above for every design element that needs further decomposition.

4. System Analysis and Design

4.1 System overview

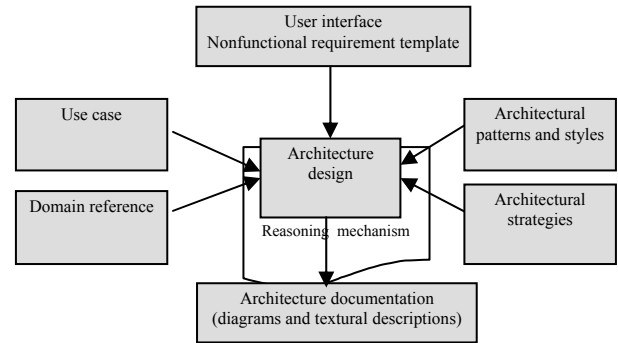


Figure 1 System structure

The system structure is shown as Figure 1. The user inputs non-functional requirements as well as other information (like domain, quality attribute), which are necessary for software architecture design, through the user interface. The user needs to specify the domain to which the system applied, and quality attributes (performance, modifiability, security, usability, testability, and availability). For each quality attribute the user chooses, the priority must be set, because some quality attributes can be conflicting with each other. And then according to each quality attribute, the user inputs corresponding non-functional requirements followed by quality general scenarios (see the Section 2.2). Domain reference, architectural patterns and styles, and architectural strategies are stored in the database as the knowledge base to support the architecture design. Architecture design part is the core intelligent processing part of the whole system. Behind this part is reasoning mechanism. As ADD method described in the Section 3, architecture drivers are shaped from non-functional requirements which are input through user interface, and the architecture design part will select the appropriate architecture strategies, which could be implemented by some architecture patterns or styles, to satisfy the architecture drivers. Functionalities from the use case are allocated to the architecture pattern. Finally a conceptual

architecture [4] with textual description will be the output.

4.2 Knowledge representation

Data is defined as a sequence of quantified or quantifiable symbols. Information is about taking data and putting it into a meaningful pattern. Knowledge is the ability to use that information. How to make software systems aware of something and how to store knowledge in computers are the core problems around knowledge representation [7]. In our system, we need to build a knowledge base which collects the real world experience for architecture design. These real world experiences include architecture strategies, architecture patterns and styles, and domain reference. The knowledge is represented in the relational form, i.e., record of information about an item, with each record containing a set of fields defining attributes and values.

Table 2 Domain reference

Application domain	Domain reference
--------------------	------------------

Table 3 General strategies for quality attributes

Quality attribute	Strategy category	Strategies
-------------------	-------------------	------------

Table 4 Specified strategies

Quality attribute	Strategies	Model	Application domain	Explanation	Pseudo code or UML diagram

Table 2 is used to store domain reference, Table 3 is used to store architectural strategies for the corresponding quality attribute, and Table 4 is used to store detailed strategy implementation in the specified application domain. Table 2 only stores some general architectural strategies, which may be applied to any application domain, but they may not be readily implemented. To give more practical strategies to the user, the general one must be specified. Table 5 is the examples for Performance strategies.

For the general strategy - maintain multiple copies, there is one pattern named Page cache pattern in the Web domain to implement this strategy, which is shown in the Figure 2. At the detailed strategy level, the user needs to understand what this strategy is and how to use it, so the Explanation item in the template will help the user to understand under what context the strategy can be used, and the Pseudo code or UML diagram will guide the user to use this strategy.

Table 5 Performance strategies

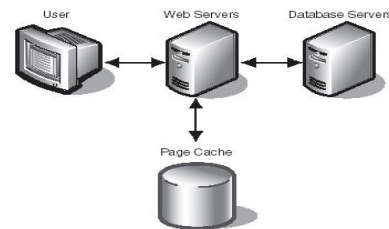
performance	Resource Demand	Increase computation efficiency
		Reduce computational overhead
		Manage event rate
		Control frequency of sampling
	Resource Management	Introduce concurrency
		Maintain multiple copies
		Increase available resources
	Resource Arbitration	Scheduling policy

Page cache pattern

Quality: Performance

Tactics: Page cache pattern

Model:



Application domain: web-based systems

Explanation:

You are working with a Web-based application that presents dynamic information to users. You have observed that many users access a specific page without the dynamic information changing. How can you improve the response time of dynamically generated Web pages that are requested frequently but consume a large amount of system resources to construct? Generating a dynamic Web page consumes a variety of system resources.

Pseudo code or UML diagram:

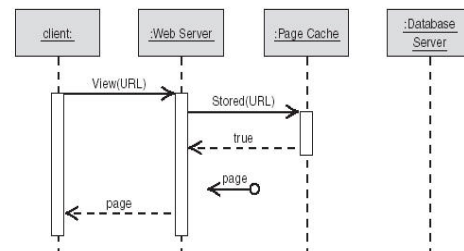


Figure 2 Page cache pattern

4.3 Reasoning mechanism

The architecture patterns and strategies are already stored in the knowledge base. Reasoning mechanism is responsible for choosing the appropriate strategies according to the user inputs. For the architectural pattern selection, architectural patterns cannot be selected only by quality attribute, because the pattern is also context-oriented. For example, if we want the system support changeability and reusability, and also the system shall flexibly interact with various users, from the Table 6 and Table 7, we can see MVC pattern should be a proper one.

Table 6 Correlation between patterns and quality attributes
 ("+" support; "-" side effect)

	Layer	Pipe and filter	Broker	MVC	...
Changeability	+		+	+	...
Interoperability			+		...
Performance	-	+	-	-	...
Reliability			-		...
Testability			+		...
Reusability	+	+	+	+	...
Portability			+		...

Table 7 Pattern description

Patterns	Context
Layer	A large system that requires decomposition
Pipe and filter	Processing data streams
Broker	Your environment is a distributed and possibly heterogeneous system with independent cooperating components
MVC	Interactive application with a flexible human-computer interface
...	...

From the Section 2, we know each quality attribute is associated with some general scenarios, when we make these general scenarios system specific, they become concrete non-functional requirements. If we know the relationship between general scenarios and architectural strategies, and for any user-input non-functional requirement we can classify it as one form of general scenario, then we will have mapping mechanism between the user-input non-functional requirements and architectural strategies.

For the relationship between general scenarios and architectural strategies, it has the following possibilities:

1. the strategy strongly contributes to this general scenario
2. the strategy contributes to this general scenario
3. the strategy has no affect on this general scenario
4. the strategy introduces side effects to this general scenario

We will ask several domain experts in this area to give weights of relationship between general scenarios and architectural strategies based on the above four possibilities. Dempster-Shafer theory will be used to combine the evidence.

5. CONCLUSIONS

Software architecture design is a task based on various experiences and specialized expertise, and at each step, stakeholders, designers and domain expert will meet together to make decision and decide what strategies they will take. The system described in this paper can partially automate this process followed by Attribute Driven Design method. A knowledge base of real world architecture experiences is being developed to support this intelligent system. Knowledge representation and reasoning mechanism has been discussed in this paper. The system will be able to reduce effort for novice software designer, reduce the total development effort and improve the time-to-market.

Acknowledgements

The authors would like to thank the reviewers and colleagues for their valuable comments and suggestions to this work.

References

- [1] Rick Kazman, "Tool Support for Architecture analysis and Design," Proceedings of the ACM SIGSOFT '96 Workshops, pp. 94-97, San Francisco, CA, October 1996.
- [2] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice (2nd edition)," Addison-Wesley, 2003.
- [3] Lawrence Chung, Brian A. Nixon & Eric Yu, "Using Non-Functional Requirements to Systematically Select Among Alternatives in Architecture Design," Proc. 1st Int. Workshop on Architectures for Software Systems, 1994.
- [4] Hofmeister, C., Nord, R., Soni, D., "Applied Software Architecture," Addison Wesley, 2000.
- [5] Len Bass, Mark Klein, Felix Bachmann, "Quality Attribute Design Primitives and the Attribute Driven Design Method," Bilbao, Spain, October 2001.
- [6] Len Bass, Mark Klein, Felix Bachmann, "Quality Attribute Design Primitives," (CMU/SEI-2000-TN-017).
- [7] Bigus, J.P., Bigus, J. (2001), *Constructing Intelligent Agents Using Java* (2nd Edition), Wiley Pub Co., ISBN: 0-471-39601-X.