5th SASTech
Iran |Mashhad
May 12 - 17 | 2011
5th Symposium on Advances in Science & Technology

Improving Software Quality by Pattern Driven Software Architecture

Ehsan Ataie, Marzieh Babaeian, Fatemeh Aghaei
Department of Computer Sciences
University of Mazandaran
{ataie, m.babaeian, f.aghaei}@umz.ac.ir
Paper Reference Number: 0902-1087
Name of the Presenter: Marzieh Babaeian

Abstract

This paper discusses about improving quality of software by taking into account non-functional requirements. Here, we present a Pattern Driven Software Architecture method which uses architectural patterns. The relationship and semantic of patterns are represented by Feature Model and Role-Based Modeling Language. Using a recursive decomposition process, at each stage in the decomposition, 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 patterns.

Key words: Architecture, Quality, Functional, Non-Functional, Requirement

1. Introduction

In creating software architectures, we have to build a structure that supports the functionality or services required for the system (functional requirements) with respect to the system qualities (non-functional requirements). In general, functional requirements define what a system is supposed to do whereas non-functional requirements define how a system is supposed to be. The International Organization for Standardization (ISO) defines quality as a characteristic that a product or service must have [http://www.praxiom.com]. Dropping quality requirements out of the software architecture design process may mean that a large amount of resources has been put into building a system that does not meet its quality requirements [Bosch (2000)]. This wastes time and money and produces an architecture of poor quality.

In this paper two main approaches that describe quality driven software architectures are reviewed and differences and similarities of them are discussed. Based on these approaches, we present a new approach called Pattern Driven Software Architecture (PDSA) which can drastically improve software quality. The remainder of this paper is organized as follows: Section 2 gives a detailed analysis of first and second techniques. In section 3, these approaches are compared and their advantages and disadvantages are described. The PDSA approach is also presented in this section. Finally, in section 4 we discuss about our current and future works.

2. Research Methodology

2.1. Analysis of First Approach

2.1.1. Background

First approach presents a quality-driven approach to embodying non-functional requirements (NFRs) into software architecture using architectural tactics [Kim (2009)].

2.1.2. The Concepts

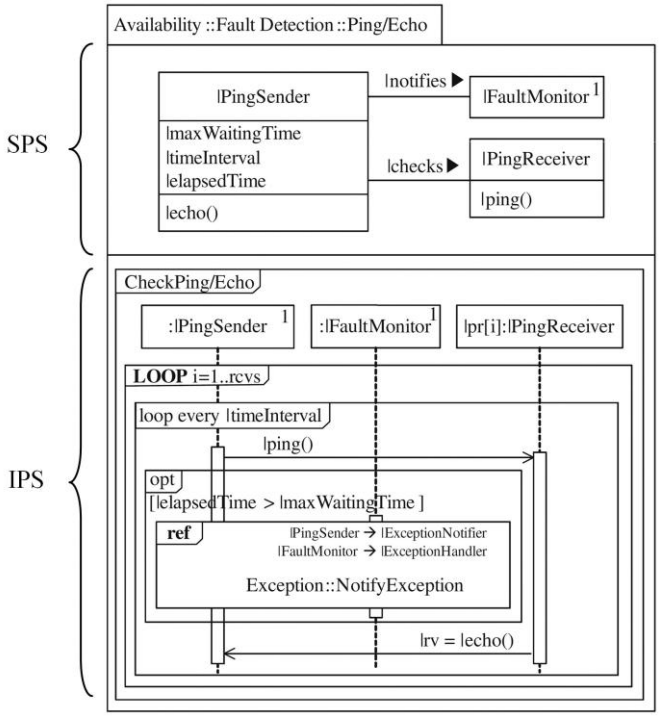
Quality attributes are non-functional requirements used to evaluate the performance of a system. An architectural tactic is a Fine-grained reusable architectural building block that provides an architectural solution built from experience to help to achieve a quality attribute [Kim (2009)]. There are many architectural tactics [Bachmann (2002)], [Bass (2003)], [Ramachandran (2002)] for various quality attributes such as availability, performance, security, modifiability, usability and testability. For every individual quality attribute a set of tactics can be defined. The relationship between quality attributes and the tactics can be shown in feature models [Czarnecki (2000)]. The coarse grained tactics can break into small grained ones. A specific notation for modeling tactics is Role Based Meta modeling Language (RBML) [France (2004)], [Kim (2007)] which is a UML-based pattern specification language. The feature models and RBML specifications are developed based on the works in [Bachmann (2002)], [Bass (2003)], [Ramachandran (2002)], [Cole (2005)], [Gagne (2005)]. Every tactic is introduced in RBML with two parts, called Structural Pattern Specification (SPS) and Interaction Pattern Specification (IPS). The SPS characterizes the structural aspects of an architectural tactic in a class diagram view and the IPS defines the interaction view of tactics. In the following, we explain some kinds of availability tactics which are introduced in this approach.

Availability is the degree to which an application is available with the expected functionality [Kim (2009)]. The *Fault Detection* is an availability tactic which is concerned with detecting a fault and notifying the fault to a monitoring component or the system administrator [Bass (2003)], [Schmidt (2006)]. This tactic can be refined into: *Ping/Echo* and *Heartbeat*. The *Ping/echo* tactic detects a fault by sending ping messages to receivers regularly. If a receiver does not respond to the sender within a certain time period, the receiver is considered to be failed. The *Heartbeat* tactic detects a fault by listening to heartbeat messages from monitored components periodically. A sender sends a heartbeat message to all the receivers every specified time interval. The receivers update the current time when the message is received. If the message is not received within a set time, the monitored component is considered to be unavailable. Fig. 1 shows the SPS and IPS of the *Ping/echo* and *Heartbeat* tactics. The *Exception* tactic is used for recognizing and handling faults. The *Exception* tactic is usually used together with the Ping/echo tactic and heartbeat tactic for handling faults. Like availability, performance and security are other non-functional requirements that contain various tactics.

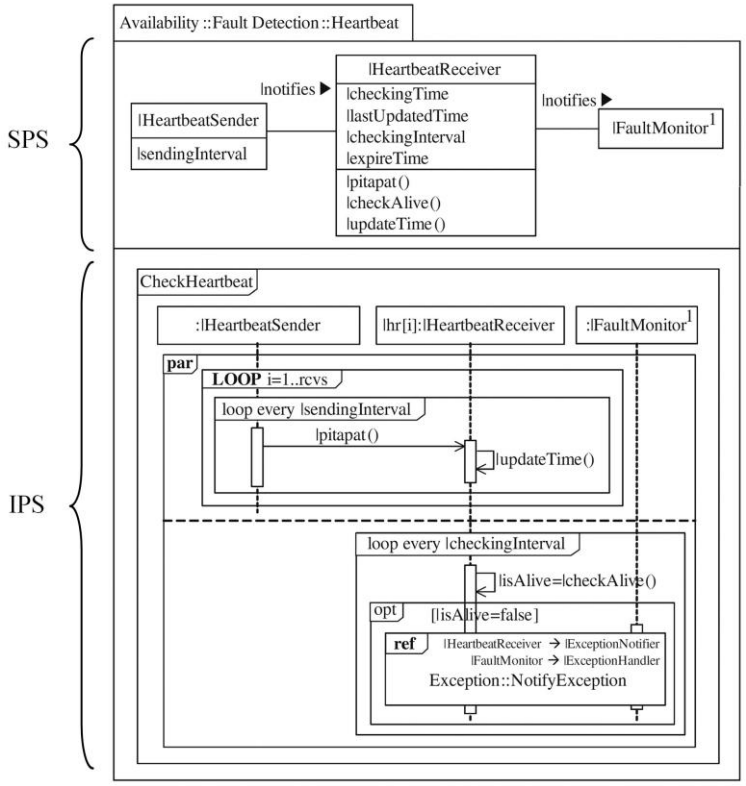
2.1.3. The Roadmap

To design the architecture of a system, the first step is to define non-functional requirements. Then, the authors described how each of the NFRs can be embodied into architecture using architectural tactic. Each NFR of the system is related to one quality attribute, and a set of tactics have to be included to obtain the quality attribute. The selected architectural tactics are composed to produce a composed tactic that exhibits the solutions of the selected tactics. As an example, Fig. 2 shows the composition of *Ping/echo* and *Heartbeat* tactics. The composed tactic is then consolidated to create an initial architecture of the application. Different instantiation of proposed generic initial architecture can be built. There is always the possibility that the different types of quality

attributes may hinder each other (e.g., security and performance may have conflicts). This approach doesn't introduce an obvious way to solve the conflict problem between quality attributes but a trade-off analysis of the two is proposed to minimize the conflicts. Finally, an architecture has been built, based on the desirable NFRs, but to add the functional requirements (FRs), we may need to make some changes in the architecture; for instance, by adding new classes, new methods and new relations between the classes.



(a) The Ping/Echo tactic.



(b) The Heartbeat tactic.

Fig 1: The Ping/Echo and Heartbeat tactics.

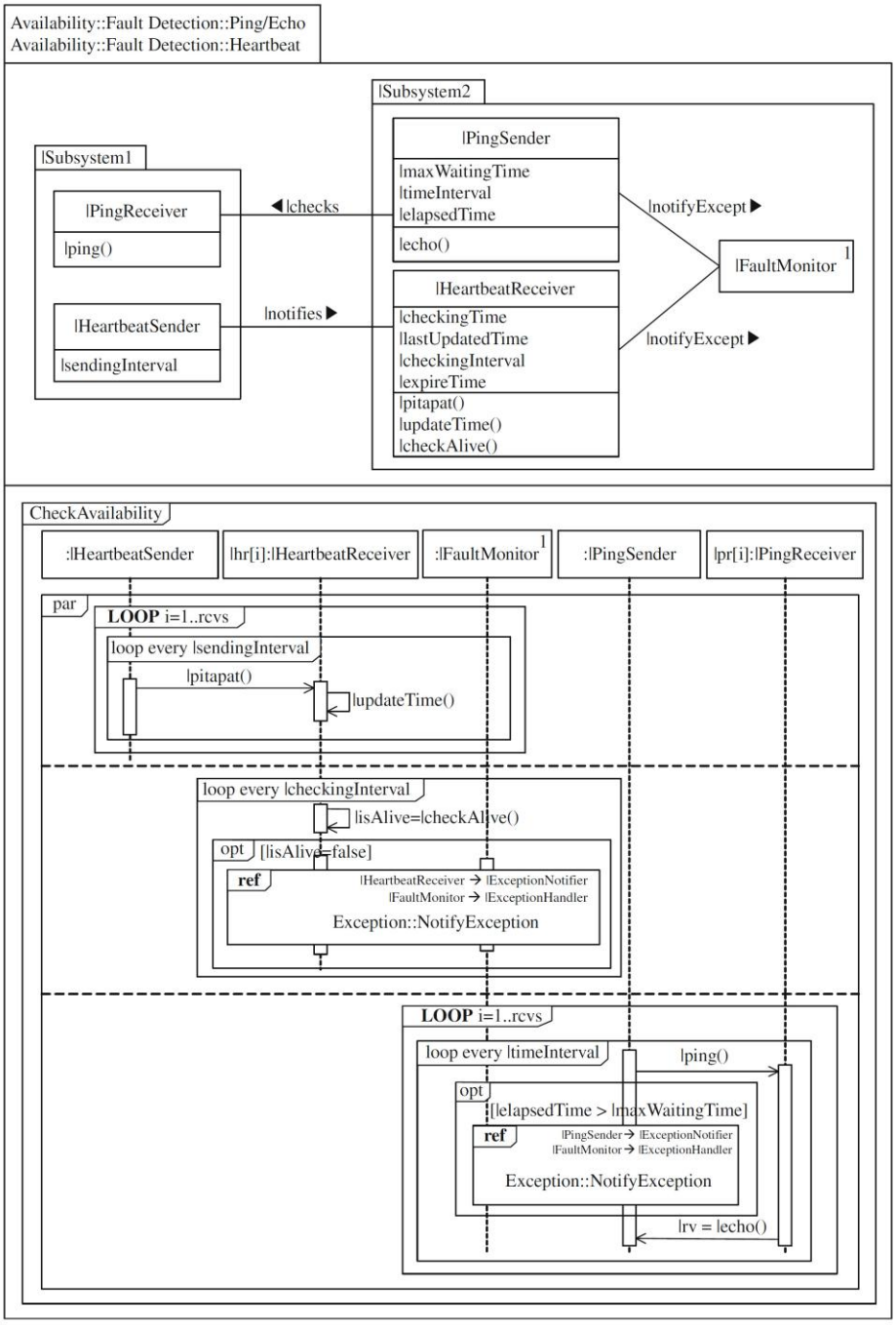


Fig 2: The composition of the Ping/Echo and Heartbeat tactics.

2.1.4. Tool support

This approach demonstrates tool support for automatic instantiation of architectural tactics [Kim (2009)]. The RBML Pattern Instantiator (RBML-PI) which is developed as an add-in component to IBM Rational Rose [Kim (2005)] is the tool this approach uses.

2.2. Analysis of Second Approach

2.2.1. Background

Second approach presents an approach for characterizing quality attributes and capturing architectural patterns that are used to achieve these attributes. For each pattern, it's

important not only how the pattern achieves a quality attribute goal but also what impact the pattern has on other attributes [Bachmann (2000)].

2.2.2. The Concepts

In this approach, the concept of general scenario and attribute primitive is introduced and a design method is described to utilize these concepts for designing a software architecture. A general scenario consists of: The stimuli that requires the architecture to respond, the source of the stimuli, the context within which the stimuli occurs, the type of system elements involved in the response, possible responses and the measures used to characterize the architecture's response [Bachmann (2000)].

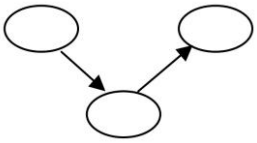
For each quality attribute one or more general scenario is introduced. An attribute primitive is a collection of components and connectors that (1) collaborate to achieve some quality attribute goal (expressed as general scenario), (2) is minimal with respect to the achievement of those goals [Booch (1996)]. A data router is an example of an attribute primitive. The data router protects producers from additions and changes to consumers and vice versa by limiting the knowledge that producers and consumers have of each other. This contributes to modifiability. The Attribute Driven Design (ADD) method is a recursive decomposition process where, at each stage in the decomposition, attribute primitives are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the component and connector types provided by the primitives [Bachmann (2000)].

2.2.3. The Roadmap

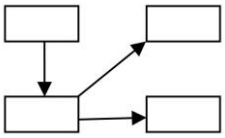
The ADD's place in the life cycle is after the requirement analysis phase. The ADD input is a set of requirements. The ADD output is a conceptual architecture [Hofmeister (2000)]. The conceptual architecture is the first articulation of architecture during the design process and therefore coarse grained. When all of the architectural drivers of the system are understood the beginning of ADD is prepared. Architectural drivers are defined as non-functional and functional requirements that are architecturally significant to the system. The first step of ADD is to choose the first design element. Then steps (b) to (e) should be repeated for every design element that needs further decomposition. In the Following, we explain each step.

a) *Choose Design Element:* In this step, we choose the design element to decompose. The decomposition usually starts with "the system" element, and is then decomposed into "conceptual subsystems" and those get decomposed into "conceptual components". The decomposition results in a tree of parents and children.

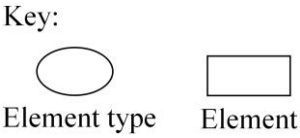
b) *Choose the Architectural Drivers:* This step determines what is important for this decomposition. The architectural drivers are usually in conflict with each other, therefore there should be a small number of architectural drivers. The quality architectural drivers determine the style of the architecture while the functional architectural drivers determine the instances of the element types defined by that style. For example we may have chosen the attribute primitive "Data Router" to support modifiability. This attribute primitive defines element types of "producer", "consumer" and the "data Router" itself. By looking at the functional drivers we may define a sensor application that produces data value, and a guidance as well as diagnosis application consuming the data value. Therefore, the functional drivers instantiate the element type "producer" into a "sensor" element and the element type "consumer" into a "guidance" and "diagnosis" element. Fig. 3 shows an instantiation of an attribute primitive.



(a) Attribute primitive, Data Router



(b) Instantiation of "Data Router" attribute primitive



(c) Element and element type

Fig 3: Instantiation of an attribute primitive

c) *Choose the Attribute Primitives:* This step chooses the Attribute primitives and children design element types to satisfy the architectural drivers. This step is designed to satisfy the quality requirements. The selection of attribute primitives is based on two factors. The first factor is the drivers themselves and the second one is the side effects an attribute primitive has on other qualities. Consider the situation where both modifiability and performance are architectural drivers. One attribute primitive for modifiability is Virtual Machine and one attribute primitive for performance is Cyclic Executive Scheduling. A Virtual Machine introduces additional checks at the interface of the virtual machine that is an obstacle for achieving performance and on the other hand, a Cyclic Executive delivers real-time performance. This is a barrier for achieving modifiability because functions have to run in a specific sequence. So clearly, the both attribute primitives cannot be used together without any restrictions because they neutralize each other. This approach attempts to solve this problem by dividing the system into two parts. The performance critical (assuming performance is more important) part and the rest. So the Cyclic Executive Scheduler would be used for the critical part and the Virtual Machine is used for the uncritical part.

d) *Instantiate Design Elements and Allocate Functionality Using Multiple Views:* In this step functional requirements are satisfied. As an example in previous step, we explained that the system has to be divided into two parts and the virtual machine is used for the performance irrelevant portion. In practice, most concrete systems have more than one application; One application for each group of functionality.

Applying functional architectural drivers, we may find that we have two different performance critical parts, such as reading and computing sensor input and keeping a radar display current. It may also be discovered that on the performance irrelevant side there should be several separate applications like diagnosis, administration and help system.

After grouping, following steps can assure us that the system can deliver the desired functionality: (1) Assigning the functional requirements of the parent element to its children by defining responsibilities of the children elements, (2) The discovery of

necessary information exchange that creates a producer/consumer relationship between those elements, and (3) Finding the interactions between the element types of the primitives that make specific patterns that can mean things like: "calls", "subscribes to", "notifies", etc. [Bachmann (2000)].
Since software architecture cannot be described in a simple one-dimensional fashion, authors used some architectural views that help focus on different aspects of the conceptual architecture. They suggested using the following views: (1) The module view, shows the structural elements and their relations, (2) The concurrency view, shows the concurrency in the system, and (3) The deployment view, shows the deployment of functionality onto the execution hardware.
The last part of this step indicates that analyzing and documenting the decomposition in terms of structure (module view), dynamism (concurrency view), and run-time (deployment view) uncovered those aspects for the children design elements, which should be documented in their interface. An interface of a design element shows the services and properties provided and required. It documents what others can use and on what they can depend.

e) *Validate and Refine Use Cases and Quality Scenarios as Constraints to Children Design Elements:* The verification of the decomposition is performed by ensuring that none of the constraints, functional requirements or quality requirements can no longer be satisfied because of the design. Once the decomposition has been verified, the constraints and requirements must be themselves decomposed so that they apply to the children design elements. This step verifies that nothing important was forgotten and prepares the children design elements for further decomposition or implementation.

3. Results and Analysis

3.1. A Comparison of First and Second Approaches

In this section, we describe the similarities and the differences between first and second approaches.

3.1.1. Similarities

- There could be a conflict between quality requirements in both approaches (e.g., performance and security).
- It is possible to solve the conflict between quality requirements in both approaches. Both approaches propose to prioritize quality requirements when a conflict occurs.
- In both approaches, quality requirements are coarse grained and can be broken into smaller grains (hierarchy structure).
- Functional requirements can be added to the architecture, in both techniques.

3.1.2. Differences

- In first approach, a quality attribute decomposes into high level tactics; then these tactics decomposes into lower level tactics; this decomposition can be continued; but in second approach, a quality attribute decomposes into scenarios. Scenarios can then decompose into attribute primitives and the decomposition is finished at this step.
- The structure and behavior of the tactics are exactly specified using RBML, in first approach, whereas the structure of the attribute primitives are defined abstractly and generally in second approach. It just consists of the components and connectors, not the classes and methods.

- In first approach, there is no emphasis on functional requirements and only non-functional requirements are satisfied, but in second approach, functional requirements are satisfied as well as non-functional requirements.
- In first approach, the requirements are assumed to be in the same level, but second approach, pays more attention to essential requirements called architectural drivers.
- Whereas second approach also takes constraints into account, first approach only embodies quality attributes.
- In spite of second approach, first approach is tool supported.
- First approach, consists of modeling notations for quality requirements (i.e. feature models and RBML), whereas second approach does not support such concepts.
- As mentioned in Similarities part, FRs can be added to the architecture in both approaches; but in spite of first approach, second approach uses a systematic way to inject functional requirements.
- In second approach, the functional requirements can be broken and there can be relations between the broken parts but first approach, does not elaborate this decomposition .
- In the Similarities part, we have said that the conflict problem is solved by giving priority to the quality requirements; but solutions are different. In first approach, if some security tactics have conflict with performance quality attribute (e.g., confidentiality tactic which involves encryption and decryption overhead), and performance has more priority than security, confidentiality tactics should be discarded. In the same situation in second approach, ADD method divides system into different components. Some of them are performance critical, in which security patterns should be ignored. Others are performance irrelevant, in which security attribute primitives can be applied.

3.2. Pattern Driven Software Architecture Approach

In order to lessen the drawbacks and to fortify the advantages of both approaches, we establish a new approach called Pattern Driven Software Architecture. By *pattern*, we mean a general concept that can encompass both tactics and attribute primitives. We employ feature model form first approach to show the relationship between quality attributes and patterns. This model, enables us to unlimitedly decompose coarse-grained patterns into more fine-grained ones. Use of RBML notation allows us to precisely describe the structure and behavior of each pattern.

Since second approach uses a more systematic method, we put PDSA on the basis of ADD method. To be brief, we do not repeat the steps. But, in step (d) of PDSA, we use 4+1 architectural views including *Logical View* (which depicts structure of the model), *Process View* (which captures concurrency and synchronization aspects), *Physical or Deployment View* (which describes the mapping of functionality onto the hardware and reflects its distributed aspects), *Development or Implementation View* (which shows static organization of software in its development environment) and *Scenarios or Use-case View* (which captures the basic functional requirements for the system as a set of use-cases) [Kruchten (1995)].

Table 1 summarizes the comparison between first and second approaches from previous section and our PDSA approach.

Specification	First Approach	Second Approach	PDSA
Conflicts between NFRs	Yes	Yes	Yes
NFR Modeling notation	Yes	No	Yes
Solving the NFRs conflict	Yes	Yes	Yes
Tool support	Yes	No	Yes
NFRs hierarchy structure	Yes	Yes	Yes
Unlimited NFR decomposition	Yes	No	Yes
Adding FRs	Yes	Yes	Yes
Systematic approach for adding FRs	No	Yes	Yes
Breaking FRs and interactions between them	No	Yes	Yes
Satisfying the constraints	No	Yes	Yes
Emphasis on more important Requirements	No	Yes	Yes
Detailed specification of patterns	Yes	No	Yes

Table 1. Comparison between first and second approaches and PDSA.

4. Conclusions

In this paper we reviewed two important approaches that embody NFRs into Software Architecture and made a detailed comparison between them. Based on these techniques, we present a PDSA approach which is more flexible and efficient than both reviewed approaches.

Our next plan is to review more approaches and present a more comprehensive version of PDSA in order to gain quality-driven software architecture.

References

Bachmann, F., Bass, L., & Klein, M. (2000). *Quality attribute design primitives*. Carnegie Mellon University, December.

Bachmann, F., Bass, L., & Llein, M. (2002). Illuminating the fundemental contributors to software architetcure quality. *Technical Report, Software Engineering Institute*. Carnegie Mellon University.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in pracrics*. 2nd ed., Addison Wesley.

Booch, G. (1996). *Object solutions: managing the object-oriented project*. Addison Wesley.

Bosch, J. (2000). *Design and use of software architectures.* Addison Wesley.

Cole, E., Conley, J., & Krutz, R. (2005). *Network security bible*. John Wiley.

Czarnecki, K., & Eisenecker, U. (2000). *Generating programming: methods, tools, and applications*. Addison Wesley.

France, R., Ghosh, S., Kim, D., & Song, E. (2004). A UML-based pattern specification technique. *IEEE transaction on Software Engineering*. 193–206.

Gagne, G., Galvin, P., & Silberschatz, A. (2005). *Operating system principles*. Addison Wesley.

Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied software architecture*. Addison Wesley.

Kim, D. (2007). *The Role-based metamodeling language for specifying design patterns, design pattern formalization techniques*. 183–205.

Kim, D., & White, J. (2005). Generating UML models from pattern specification. In: the proceeding of the Third ACIS International Conference on *Software Engineering Research, Management and Applications* (SERA2005). 166–173.

Kim, S. (2009). Quality-driven architecture development using architectural tactics. *The journal of Systems and Software*. March.

Kruchten, P. (1995). Architectural Blueprints- The 4+1 view model of software architecture. *IEEE Software Magazine*. vol. 12, no. 6, 42–50.

Praxiom Research Group ISO 9000 Definitions, http://www.praxiom.com.

Ramachandran, J. (2002). *Designing security architecture solutions*. John Wiley.

Schmidt, K. (2006). *High availability and disaster recovery: concepts, design, implementation*.