Pattern-Driven Architectural Partitioning

Balancing Functional and Non-functional Requirements

Neil Harrison
Computing and Networking Sciences Department
Utah Valley State College
Orem, Utah, USA
harrisne@uvsc.edu

Paris Avgeriou
Department of Mathematics and Computing Science
University of Groningen
Groningen, the Netherlands
paris@cs.rug.nl

Abstract— **One of the vexing challenges of software architecture is the problem of satisfying the functional specifications of the system to be created while at the same time meeting its non-functional needs. In this work we focus on the early stages of the software architecture process, when initial high-level system partitioning is being performed. Specifically, we study the problem of system partitioning with respect to both functional requirements and quality attributes. Architecture patterns are particularly well-suited to simultaneously addressing functional requirements and quality attributes. They support architects in considering both, understanding the impact of decisions on other attributes, and making tradeoffs among them. Existing architectural design methods accommodate pattern use, but do not exploit it in detail. We propose a pattern-based approach that leverages the benefits of patterns, and fits well with existing methods.**

Keywords- software architecture; architecture design; system partitioning; quality attributes; architecture patterns

I. INTRODUCTION

During the process of designing the architecture of a system, the architect is faced with the challenge of proposing solutions that can be implemented within the constraints of the project, while satisfying a set of system requirements. A significant part of the early phases of the architecture process is making decisions about how the system should be broken into logical partitions, in order to facilitate understanding, implementation, and extension of the system. Partitioning the system can be particularly daunting because the architect needs to cope with both functional and non-functional requirements.

Functional and non-functional requirements are fundamentally different in nature. Functional requirements describe required behavior of the system in terms of required activities [15]. They state what the system should do [16]. On the other hand, non-functional requirements describe some quality characteristic that the system must have, such as response time, ease of use, high reliability, or low maintenance costs [15]. In the field of Software Architecture, non-functional requirements are usually called quality attributes [3]. Functionality and quality attributes are orthogonal [4].

Functional requirements and quality attributes are often specified differently. Functional requirements can usually be concretely specified, e.g., with user scenarios or use cases [8]. Quality attributes are specified as system-wide characteristics. The non-functional aspects are often critical; for example, system reliability or usability may be the key differentiators among products with similar feature sets.

While functional aspects of a system can often be readily decomposed into different functional modules, non-functional aspects of a system usually cannot. On the contrary, they tend to span the entire system; all modules of a system must posses the required non-functional aspects of the system. Furthermore, functional and non-functional aspects of a system are interconnected: decisions made in order to facilitate implementation of some functional features may be compatible or incompatible with the desired non-functional attributes of the system.

A key activity of software architecture is the decomposition or partitioning of the system into architectural elements [14]. This tends to be done logically according to the functions that the system is to provide (i.e., what the system should do.) [5]. However, architects cannot focus exclusively on the functional requirements, but must consider the quality attributes as well. While the quality attributes are not implemented in single partitions, the decisions made about the partitioning of the system can affect the ease with which certain quality attributes can be satisfied. For example, security can be made easier or more difficult to implement by the architectural partitioning selected – separation of the system into layers enables security to be implemented easily at a certain layer. On the other hand, layers of components often require high messaging overhead between individual layers, and thus hurt performance in many cases. An architect considering a layered architecture is thus confronted with making tradeoffs between improved security and degraded performance. This tradeoff decision must be made early in the architecture process; when the initial gross partitioning is being considered.

In this paper we study how to perform a software system partitioning while trying to satisfy both functional requirements and quality attributes. We learned that architects consider both simultaneously, and use architecture patterns to help them do so. The application of patterns offers a reusable and proven way to partition a system with known consequences to the quality attributes. In order to generalize this outcome and provide a systematic, disciplined way of system partitioning, we propose a new approach, called Pattern-Driven

Second International Conference on Digital Telecommunications (ICDT'07)
0-7695-2910-0/07 $20.00 © 2007 IEEE

Architectural Partitioning. This approach integrates well with common architectural design methods. Significantly, many such methods mention the use of architecture patterns, but they do not explore in depth how the patterns can be used, or the significant benefits that accompany their use.

The rest of the paper is structured as follows: Section II describes the problems of system partitioning and balancing functional requirements and quality attributes. Section III presents an experiment of architectural partitioning that demonstrated the use of patterns. Section IV proposes the pattern-driven architectural partitioning approach while section V places it in the context of existing architecting methods. The paper finishes with a future work section.

II. THEORETICAL BACKGROUND

A. System Partitioning

Early in the architecture process, decisions are made about the general direction and partitioning of the system. These decisions are of necessity made with incomplete information about the system. Nevertheless, the decisions made at this early time can have significant long-term impact on the system. For example, decisions can affect the ease with which features are implemented and tested, and later features added. And as stated, they can impact the ability of the system to fulfill its quality attributes.

Practice has shown that implementations of quality attributes tend to cut across all partitions of a system. For example, in order to meet reliability requirements, each developer may need to follow a prescribed approach to error handling. This may include such rules as these:

- Check that a pointer or array index is valid before using it.
- Check that the arguments supplied to a function are valid.
- Check a function's result to see if it failed, and handle any failure as gracefully as possible.
- Check that a message arriving on an external interface is properly formatted.
- Start a timer when sending a message that could lead to a hung resource if the destination does not respond [17].

In addition, the architect must consider at least the following:

1. how to partition the system into modules that can be readily implemented by the staff of the project
2. how to satisfy the functional requirements of the system through this partitioning
3. the influence of the partitioning and the system's quality attributes on each other

These three considerations are not independent of each other, thus one must not neglect any of them during the course of designing the architecture. Thus a crucial part of software architecture is the balancing of functional requirements and quality attributes, and making appropriate tradeoffs among them.

B. Balancing Functional Requirements and Quality Attributes

While it is clear that architects must consider both functional requirements and quality attributes, how they do so is less clear. One might consider two general approaches:

The architect may consider the functional requirements, and partition the system accordingly. Then apply suitable changes to accommodate the quality attributes. This approach is proposed in the QASAR method [5].

The second approach is to consider both functional requirements and quality attributes simultaneously as the system is partitioned. This approach is either embraced or accommodated by most current architecture methods, as will be shown in the related work section. In our experience, this is the approach followed by most architects.

Functional requirements and quality attributes may be at odds with each other. In particular, architectural decisions that accommodate a functional requirement may conflict with a quality attribute, or vice versa. The architect must be aware of such side effects and make balancing decisions.

Using architecture patterns helps to alleviate this problem to a certain extent. Architecture patterns have visible partitioning which addresses functional requirements. In addition, they describe their impact on quality attributes. Therefore, as an architect uses an architecture pattern, he or she also gets knowledge of the non-functional impact on the system. Architecture patterns provide a convenient and powerful way to address both functional requirements and quality attributes of a system, as explained below.

C. Advantages of Using Architecture Patterns

There are at least four key benefits to using architecture patterns during architectural design:

First, Architecture patterns show singular power in linking functional and non-functional consequences of an architectural approach together. The solutions have been verified through extensive experience, which leads to understanding of their impact on non-functional aspects of systems. Patterns include context (preconditions for the use of the pattern), forces (considerations that may make the system especially difficult to design), and resulting context (important consequences of the solution). This additional information associated with a pattern is a powerful source of knowledge about a proposed architecture. For example, the Model-View-Controller pattern in [6] describes this benefit: "… multiple views [of the model] may be open at the same time, and views can be opened and closed dynamically." This shows a positive impact on the quality attribute, usability. It also notes the following liability: "Controller and view are separate but closely-related components, which hinders their individual reuse." This shows that the pattern can have a negative impact on the quality attribute, reusability. This pattern includes numerous other benefits and liabilities to various quality attributes. An architect

can use this information found in architecture patterns to help make decisions about how to design the system.

Second, architecture patterns can also help architects identify when an architectural approach might introduce conflicting approaches to different quality attributes. For example, a system may have both performance and security requirements. The architect may wish to use the Pipes and Filters pattern to facilitate performance gains through parallel processing [6]. However, the Pipes and Filters pattern is weak in supporting security. This additional information may cause the architect to carefully consider the usage of this pattern before it becomes too late.

Third, patterns have rich relationships among them. These relationships may include the following: A pattern may influence the use of other patterns, a pattern may specialize the use of another, two or more patterns may be alternatives, etc. This helps when an architect is trying to reason about the consequences of a combination of patterns upon the quality attributes. It also helps the architect to find alternative patterns as solutions to the same problem.

Fourth, patterns can be used as part of the natural flow of architectural design. For example, architects may be reluctant to interrupt the creative flow of architecture to document architectural decisions made. The constraints of real-world industrial projects do not allow architects the luxury of fully exploring the problem space or documenting the different dimensions of the project's architectural knowledge. However, patterns tend to emerge naturally during the course of architecture, and their use can be easily documented without disrupting the architectural design process. Architects can use them almost naturally within the context of almost any architecture process they use.

III. A Study of Architecture Pattern Usage

In order to observe how architects partition a system and tackle functional and quality aspects, we conducted a laboratory architectural exercise. The aim was not to perform a sophisticated empirical software engineering experiment but a simple, intuitive exercise in architecting. In this exercise, two teams of six members each were given a set of requirements for a web-based e-commerce system, and were asked to design a preliminary partitioning for the system. We observed each team in action, and noted what things they discussed, and the architectural decisions they made. The teams had about three hours in which to work, so they knew that they could only perform an initial partitioning.

The teams differed in experience level. One team consisted almost exclusively of highly experienced software architects; all had over ten years of software development experience, and most had over five years of software architecture experience. On the other hand, the second team had relatively inexperienced people. While all had development experience, few had any architecture experience. It was instructive to compare the processes of these teams with each other.

The requirements the teams received were about one page in length, and consisted of a short description of the system as well as a set of requirements. The requirements included both functional requirements and quality attributes. The requirements were not grouped as such; i.e., there was not a separate section for quality attributes. By doing so, we did not call attention to the quality attributes.

A. Results

We observed both groups, and noted how they considered the requirements. Both groups began by identifying reliability as the dominant quality attribute of the system. In addition, they identified security, performance, and scalability as other significant quality attributes. At this time, neither group made major decisions about approaches to reliability the other quality attributes, but rather discussed the requirements to gain clarification about what they meant.

The groups then turned their attention to the major functional aspects of the system, the communication architecture. They continued to frequently discuss all the quality attributes as they discussed the functional aspects of the system.

After some time, both groups began to converge on the same general architecture, following the Broker architecture pattern [6]. Neither group explicitly identified the Broker pattern and then sought to apply it to the problem. Rather, the overall shape of the Broker pattern emerged, and the team recognized it as the pattern. Both groups noted how the pattern satisfied the functional requirements as well as the reliability requirement, but that they would need such features as server duplication supported by the Broker pattern, to achieve high availability. At the same time, the Layers pattern emerged, and was briefly considered. The experienced team, with more extensive exposure to architecture patterns, noted that they intended to consider patterns more explicitly given more time.

B. Interpretation

Our hypothesis was that early in the architecture process, architects use architecture patterns to select an initial partitioning of the system to be built. Furthermore we believed that architects do not neglect the quality attributes, but they consider them in parallel to the functional aspects of the system. Their use of the patterns means that they get the attendant non-functional behavior "for free," as a by-product of the pattern usage.

We observed that at least for certain quality attributes, architects consider them at the same time as they consider the functional requirements. In fact, they work them together, bouncing from one to the other rather quickly. This indicates that they fully appreciate the interaction between functional requirements and quality attributes – that an architectural solution must address both at the same time.

The architecting exercise showed that in order for architectural methods to be most effective, they must support the consideration and documentation of functional requirements and quality attributes more or less simultaneously. At the very least, they must allow it to happen. For example, many of the methods and tools for documenting architectural decisions allow simultaneous consideration and documentation of functional and non-functional decisions by

allowing both types of decisions to be documented in the same way. The next section describes our own approach to tackle this issue.

IV. Pattern-Driven Architectural Partitioning

As demonstrated by the experiment results, architecture patterns fit well into the initial stages of the architecture process, as architects analyze the system's quality attributes and propose overall approaches to the system partitioning. For this purpose we have defined an approach, called Pattern-Driven Architectural Partitioning, comprised of the following steps:

1) Identify the most prominent architectural drivers of the system. These include both functional requirements and non-functional quality attributes, and they are prioritized according to the stakeholders concerns.

2) Select candidate architecture patterns that address the needs of the architectural drivers. This is done by examining the context and the problem statements of the patterns to find possible matches, and the consequences to study the estimated impact of the patterns' solution on the system functionality and quality attributes. Some concerns may be adressed by the same pattern, and conversely, some patterns may adress the same concern. Each concern should be addressed by at least one pattern. Note that this is more of a qualititive matching rather than an architetural evaluation, that follows in a later stage.

3) Partition the system by applying a combination of the candidate patterns. Patterns propose solutions, where architectural elements play specific roles and have specific relationships. In the combination of the patterns, some elements may need to play more than one role and some relationships may be combined. A single system partitioning must emerge by combining the patterns.

4) Evaluate whether the partitioning (specified by the selected patterns) satisfies the architectural drivers. This step may include the following activities:

a) Examine the forces of the pattern to understand any challenges posed by the solution, and determine whether they apply to the current system.

b) Examine the consequences of the patterns to determine whether the architectural drivers are satisfied.

c) Examine the consequences of the patterns to determine whether they impose constraints on the system that may require additional patterns, or may render any patterns impractical for this system. If additional patterns are needed, return to step 2 to select patterns. Additional patterns can be found in the related patterns sections of the pattern description.

d) Examine the interaction among the patterns selected to see whether the impact on the quality attributes are compatible.

5) Perform trade-off with respect to the different architectural drivers. This entails compromising some drivers in favor of others, and consequently affecting the system partitioning. If no optimal trade-off can be found, return to step 2 to select alternative patterns or substitute some patterns for others. If some architectural drivers have not been addressed yet, also return to step 2 to select additional patterns, as needed.

This process can be repeated as additional architectural drivers are identified, or if fundamental attributes of the system change. In subsequent iterations, it is normal to consider the impact of existing patterns on the newly identified drivers before adding new patterns; i.e., begin with step 2.

The results of this approach form a starting point for the architectural design. The next steps vary in different architecting methods, but they usually involve further decomposition, documentation of other views, and eventually architectural evaluation and evolution studies.

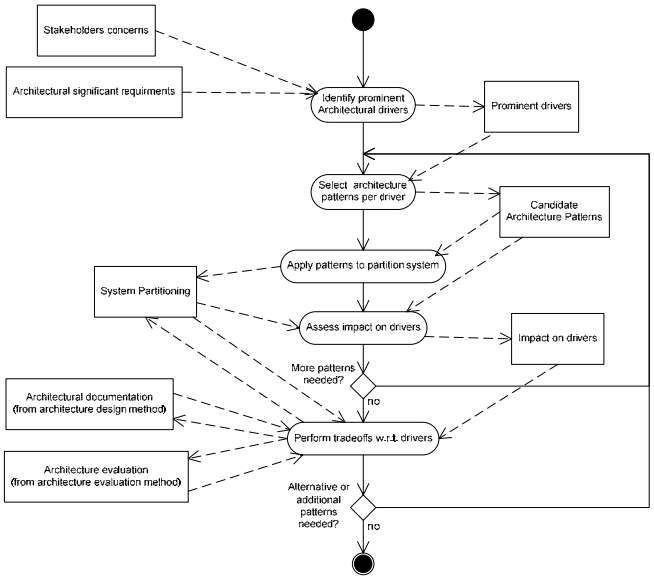

Figure 1. The Pattern-Driven Architecture Partitioning

B. Example

In order to illustrate how to use PDAP, consider the web-based purchasing system as described above. It displays items for sale and allows the user to select items to purchase. At checkout time, it accepts shipping and payment information from the user, and verifies the credit card information. Upon sale confirmation, it generates a credit card payment record, and a shipping order. The system has the following important quality attributes:

1. Reliability: the integrity of transactions must not be compromised. If the user aborts the transaction, or if there is an error that prevents the completion of the transaction, neither a bill nor a shipping order may be generated.

2. Security: the financial aspects of the transaction must be safe for the user, and the system must be safe from compromise by break-ins.

3. Scalability: as the business grows, the system must be able to easily expand to handle the increased traffic.

4. Modifiability: the business may expand to different types of products, or into different countries. Because the business growth is as yet unknown, the system must accommodate such modifications.

5. Performance: the system must be able to handle occasional traffic spikes; for example, in response to a special offering.

Step 1: We select reliability and security as the most prominent quality attributes to consider early. Of the functionality, we note that the requirement that the system provide web-based transactions is the key requirement. This gives us the most important architectural drivers of the system.

Step 2: In order to maintain transactional integrity across the web, we isolate transaction processing inside the server and put a gateway process in front of it. We see that this matches the Broker architecture pattern exactly.

Since we need strong security, we consider other patterns that could enhance security. We see that the Layers pattern supports many aspects of security, such as role-based access control.

We haven't addressed reliability in detail yet, but believe that the Broker and Layers patterns will support reliability well, so we continue to step 3.

Step 3: We follow the Broker and Layers patterns to create tentative partitions. We see that one partition encapsulates the role of the Broker, and that the server software behind it is partitioned into separate layers.

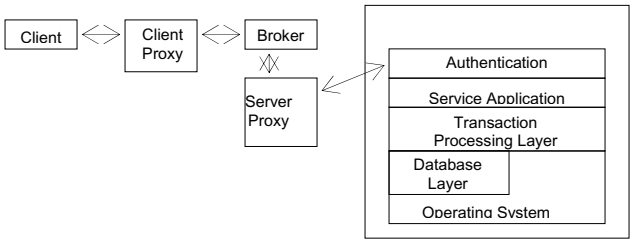

Figure 2. The Broker and Layers Patterns

Step 4: We examine the Broker pattern for its impact on other quality attributes. We see that the Broker pattern supports isolation of operations, and could easily support password-protected sessions. We see that this is compatible with the Layers architecture.

We examine the consequences of the Layers architecture, and note that one consequence is that it is not particularly good for high performance. This was not a key driver, but we consider it, and decide that high performance is not of key importance. If it were, we would evaluate whether or not to keep the Layers pattern.

We note that both Broker and Layers are compatible in their support of transactional integrity, which is a major component of reliability. However, nonstop processing (availability) is not addressed.

The Broker process could be 'choke point' during times of heavy load, hurting performance. If this is deemed an important risk, we must either modify the Broker pattern or select a different pattern. We look into performance tactics and see that duplication of the Broker process looks promising, so we select this variation, pending more detailed analysis.

Step 5: It appears that security is at least addressed, though it is not yet clear whether it is sufficiently strong. Reliability is not compromised, but nonstop processing is not addressed. We return to step 2, where we examine server duplication patterns such as Warm Standby or Hot Standby [17].

In later iterations, we consider capacity and modifiability. Before we add patterns, we consider the impact of current patterns on those drivers. We see the Broker naturally supports distribution of traffic to multiple servers, which gives strong support for scalability. This also supports reliability, in that the servers can be duplicated. However, we note that the Broker process becomes a single point of failure; a potential vulnerability in reliability. We will address that issue with other patterns, or as we consider tactics during more detailed design.

V. Related Work

Software system partitioning is only part of the whole architecting process. Numerous such processes have been proposed, and several are being actively used and studied. We highlight some of the best-known below, and describe how they support the architect in addressing both functional and non-functional aspects of systems. This will help us in setting the PDAP approach in the appropriate context.

The Attribute-Driven-Design (ADD) Method [4], consists of three major steps:

1. Choose the module to decompose.
2. Refine the module by choosing architectural drivers and a set of tactics that satisfies the architectural drivers. Verify and refine the use and quality scenarios.
3. Repeat the above steps for every module that needs further decomposition.

The quality scenarios may be either functional or non-functional; ADD does not make a distinction. However, since non-functional quality attributes span partitions, consideration of them would occur in the earliest iteration of the ADD process. The PDAP process would apply most naturally at that early iteration.

Several architecture methods center on various views of the system. The Siemens' 4 Views method [10] has four views: conceptual, execution, module and code architecture. The Rational Unified Process 4 + 1 Views [11][12] centers on four views to describe the architecture: logical view, process view, implementation view, and deployment view. The additional view is a use case view that relates to all the other views. Design strategies are proposed to address these issues, which may be architecture patterns. PDAP could help architects identify the impact of the use of those patterns.

Hofmeister et al have proposed in [9] a general model of software architecture design based on these three approaches,

as well as two others used in industry. The general model consists of three activities: Architectural analysis identifies the architecturally significant requirements (ASRs) of the system; namely those requirements that influence its architecture [13]. The second activity is architectural synthesis, the creation of candidate architectural solutions to address the ASRs. Architectural evaluation, the third activity, validates the candidate solutions against the ASRs. These activities are performed iteratively, at multiple levels of granularity. In this general view, ASRs may include quality attributes as well as functional requirements. PDAP is highly compatible with this general model. The early steps of identification of quality attributes map to architectural analysis. Selecting patterns is architectural synthesis. Performing analysis of tradeoffs of quality attributes is architectural evaluation.

Bachmann et al [3] have proposed using a reasoning framework to assist in architectural design. Reasoning frameworks are to be created for individual quality attributes, such that relevant requirements can be applied to an architecture description. Proposed architecture transformations are thus generated that will satisfy some or all of the requirements. The framework modularizes quality attribute knowledge so that it can be used inside a general design method, and reduces the problems of scale in using quality attribute models in architectural design.

The partitioning proposed in PDAP can serve as a starting point for architectural refinement through quality attribute reasoning frameworks. In addition, patterns can assist the architect in this task by showing the impact of certain architectural decisions (those embodied in patterns) on multiple architectural drivers simultaneously. This can be of significant value, since the reasoning frameworks are independent, and managing interactions among them is done by the architect.

VI. Conclusions and Future Work

One of the key challenges of software architecture is recognizing and dealing with the side effects of architectural decisions. Decisions about system partitioning are usually made according to functionality, though they have major implications on quality attributes. Because patterns have a great deal of contextual material and rich interconnections, they are a powerful tool for these challenges. PDAP is a way for architects to exploit this information, and can thus add significant value to existing architectural methods.

The most significant information needed for PDAP is the impact of architecture patterns on non-functional aspects of systems. To this end, each of the well-known architecture patterns must be analyzed, and a complete catalog of the patterns with their non-functional impacts must be created. Most of the common architecture patterns have been cataloged [2],and some already contain extensive descriptions of their non-functional consequences (for example, see [6].) This work will be most effective, however, when it is complete and available in one place. Cross-referencing capabilities, e.g., searching the catalog by quality attribute would be extremely useful. In addition, some combinations of patterns are more commonly found than others. We plan to explore these prevalent combinations, and make them reusable in the third step of PDAP.

The use of architecture patterns in conjunction with the above architectural methods can be further explored. In particular, we are interested to explore the interaction of patterns with quality attribute reasoning frameworks. This should include analysis of the interaction of multiple quality attributes within each pattern, in order to understand how to focus on a single critical quality attribute. After the impact of patterns on quality attributes is cataloged, it might be used as some of the knowledge in the reasoning frameworks.

PDAP is described in a 'green field' architecture setting, but it can be applied to legacy systems as well. Such use can be explored in more detail. We intend to validate both architecting scenarios in an industrial setting.

References

[1] P. America, H. Obbink, and E. Rommes, "Multi-View Variation Modeling for Scenario Analysis," in Proceedings of Fifth International Workshop on Product Family Engineering (PFE-5), Sienna, Italy, 2003, Springer-Verlag, pp. 44-65.

[2] Avgeriou, P. and Zdun, U.: "Architectural Patterns Revisited – a Pattern Language", 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005), Irsee, Germany (July 2005)

[3] Bachmann, F.; Bass, L.; Klein, M. & Shelton, C. Designing software architectures to achieve quality attribute requirements IEE Proceedings, 2005, vol. 152.

[4] Bass, L.; Clements, P. & Kazman, R. Software Architecture in Practice 2nd Edition Addison Wesley, 2003

[5] Bosch, J. Design and use of software architectures: adopting and evolving a product-line approach ACM Press/Addison-Wesley Publishing Co., 2000

[6] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P. & Stal, M. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns Wiley & Sons, 1996

[7] Clements, P.; Kazman, R. & Clein, M. Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley Publishing Co., 2002

[8] Cockburn, A. Writing Effective Use Cases Addison-Wesley Longman Publishing Co., Inc., 2000

[9] Hofmeister, C.; Kruchten, P.; Nord, R.L.; Obbink, H.; Ran, A. & America, P. Generalizing a Model of Software Architecture Design from Five Industrial Approaches, IEEE Computer Society, 2005, 77-88.

[10] Hofmeister, C., Nord, R., & Soni, D. Applied Software Architecture, Addison-Wesley 2000

[11] Kruchten, P.: "The 4+1 View Model of Architecture", IEEE Software 12(6) (1995)

[12] Kruchten, The Rational Unified Process: an Introduction, 3rd edition, Addison-Wesley, 2004

[13] Obbink, H, Kruchten, P., Kozaczynski, W., Hilliard, R., Ran, A., Postema, H., Lutz, D., Kazman, R., Tracz, W., & Kahane, E., Report on Softare Architecture Review and Assessment (SARA), Version 1.0, February 2002

[14] Perry, D. and Wolf, A. Foundations for the Study of Architecture, ACM Sigsoft Software Engineering Notes, vol 17 no 4, Oct. 1992

[15] Pfleeger, S. L. and Altee, J. M. Software Engineering: Theory and Practice, 3rd edition, Pearson Prentice Hall, 2006

[16] Sommerville, I. Software engineering (8th ed.) Addison Wesley Longman Publishing Co., Inc., 2007

[17] Utas, G. Robust Communications Software: Extreme Availability, Reliability and Scalability for Carier-Grade Systems, Wiley, 2005