From Requirements to Architecture: The State of the Art in Software Architecture Design

Lin Liao
Department of Computer Science and Engineering
University of Washington

Abstract

Software architecture design has become an indispensable step in large software development. Because the involvement of non-functional requirements, this task is very complex and informal. This paper presents a survey on modern methodologies of software architecture design, i.e. the pattern-based design, multiple-view model, evaluation and transformation based design, and architecture-based product lines design. A few important trends are also examined and some existing problems are discussed concisely.

1. Introduction

In contrast with the architecture, which is one of the oldest arts and could be traced back to the Great Wall in China and pyramids in Egypt thousands of years ago, software architecture is still in its nascency. Although the foundations of such a concept came into being since 1960's with the development of software, it is in the 1990's that the term "software architecture" began to attract substantial attention both from the research community and from the industry [14]. The most important motivation is the growing of software systems: hundreds of thousands of lines of code were commonplace. The challenges to create, evaluate and maintain these huge systems have greatly stimulated the growth of such a field. The importance of software architecture for large and complex software systems can be explained by the following reasons [7].

1. Mutual communications. Most, if not all, of the system's stakeholders can use software architecture as a basis to understand the system, form consensus, and communicate with each other.
2. Early design decisions. The software architecture is the earliest artifact that enables the priorities among competing concerns to be analyzed. Such concerns include the tradeoffs between performance and modifiability, between the maintainability and reliability, and between the cost the current development and the cost of the future development.
3. Transferable abstraction of a system. The model of software architecture is transferable across systems. In particular, it can be applied to other similar systems and promote large scale reuse.

As most other important concepts in computer science, no definition of software architecture is commonly agreed upon [4] [14] [20]. However, it is commonly agreed that software architecture is concerned with components of the system and their interactions. And most people agree that the main concern of software architecture is the high-level structure, in contrast with the detailed design of software.

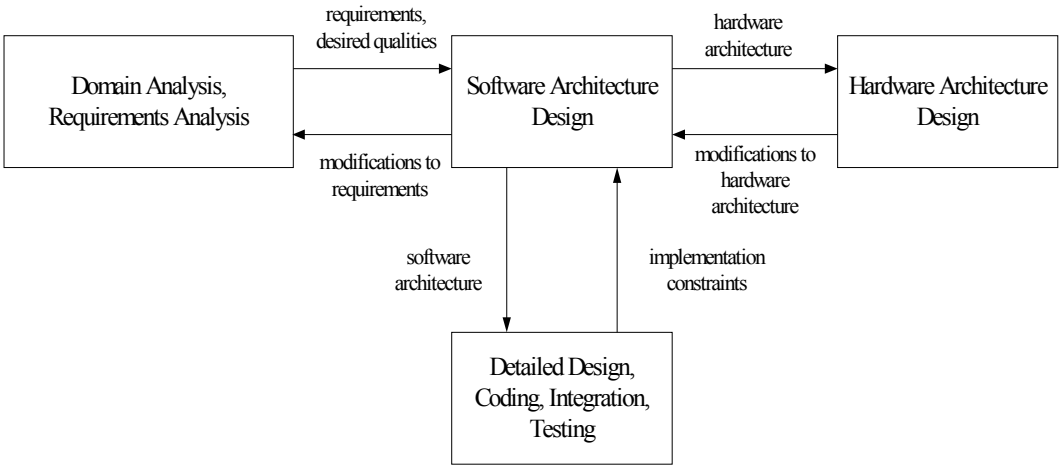

Figure 1. Relation of software architecture to other development tasks [12]

Figure 1 shows how the software architecture fits in the software development process and the interactions between tasks [12]. The first task is domain and requirement analysis and it may produce the requirement specifications. The requirements can be classified as functional requirements (FRs) and non-functional requirements (NFRs), e.g. reliability, maintainability, cost, etc. These requirements are the key input to the second task, software architecture design. As the architect reviews the requirements and proceeds with the design, some modifications to the requirements may be needed. The architect also works closely with hardware architecture team and the hardware architecture is another input to the software architecture design. The software architecture then guides the implementation of the software, including the detailed design, coding integration and testing.

To make software architecture useful in practical software development, four problems have to be addressed:

1. How to describe software architecture explicitly? A number of researchers in this area have proposed quite a few formal notations for representing and analyzing software architecture, known as architecture description language (ADL), such as Adage, Aesop, Rapide, SADL, UniCon, etc. [9].
2. How to design good architecture for software? This task becomes more challenging when a number of NFRs need to be considered. In the traditional software development process, the methodologies often existed in the minds of architects or were documented informally. It is an important goal for software architecture community to make ordinary skilled engineers to understand and use these methods.
3. How to analyze an existing architecture? How to predict whether an architecture will result in an implementation that meets the requirements? The purpose of the software architecture is not only to describe the important aspects, but also to expose them so that the architect can reason about the design.
4. How to make sure the software implementation consistent with the architecture design? In practical software development, it is commonplace that the implementation differs so much from the architecture documents that the architecture design only serves as a legacy.

These four problems are closed related to each other. Great progress has been made in these directions over the past decade. This paper is focused on the second problem, that is, given requirement specifications, both FRs and NFRs, how to design a good architecture. In practice, ideal software architecture does not exist. The good architecture here means when the system is implemented according to the architecture, it meets the requirements and resource budgets [12].

In this paper, I want to discuss some modern methodologies on architecture design. In section 2, pattern-based architecture design is introduced, followed by the multiple-view model in section 3 and evaluation and transformation based design in section 4. Then in section 5, we introduce the product lines method. And in section 6, our opinions about this area are discussed.

2. Pattern-Based Architecture Design

One of the best ways to learn how to design is to see what good designs are. The use of patterns is pervasive in many engineering disciplines. Indeed, an established, shared understanding of the common forms of design is one of the hallmarks of a mature engineering field [20].

A pattern provides significant semantic context about the kinds of concerns, the expected path of evolution, the overall computational paradigm, and the relationship between a system with other similar systems [9]. Patterns are often categorized into two levels based their scales: architecture styles and design patterns [4][8] [12] [20] (some literatures classify architecture styles further into architectural styles and architecture patterns [5]).

Shaw and Garlan [10][20] and Buschmann et al. [4][19] present a list of well-recognized patterns, such as pipes and filters, layered systems, repositories, etc. These styles are important because they differentiate classes of designs by offering experiential evidence of how each class has been used along with qualitative reasoning. Architectural styles are different from design patterns in that they affect the whole architecture or a larger part of it. Transforming an architecture by imposing an architecture style often results in a complete reorganization of the architecture [5].

The most well-known design patterns are listed in [8]. The main difference of design pattern from architectural styles is that design patterns are often only applied to one part of an architecture. In fact, some design patterns may be used to describe the interaction within an architectural element and are dealt with in the detailed design, not in the architectural design.

Each description of a pattern usually includes at least three parts: context, structure and consequences [4][8] [12] [20].

1. The context illustrates a design problem and how the class and object structures in the pattern solve the problem. It addresses questions like “What are the situations in which the pattern can be applied?” and “What are examples of poor designs that the pattern can address?”
2. The structure represents the components and their interaction textual description accompanied by some pictures and diagrams. Often some examples are provided to illustrate the solutions.
3. The consequences of a pattern illustrate how the pattern support its objectives and what are the tradeoffs in the pattern.

Such a shared repertoire of useful patterns is a great knowledge base about design principles and techniques. It makes it much easier to analyze the tradeoffs between conflicting requirements and different designs.

The idea of pattern-based architecture design is to use such a knowledge base to guide the design of software architecture. We illustrate such an approach by a simple example.

Suppose in a system, there exists a data source and different display modes are required to show the data, e.g. spreadsheets, bar charts and pie charts. The data in different views needs to be consistent and new display modes may need to be added later.

How to design such a subsystem that meets both the FRs and NFRs? If the architect is familiar with various patterns or has a list of patterns to look up, the Observer pattern (also known as Publish-Subscribe) seems to be a candidate. In the context description of Observer pattern [8], it says, “A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.” It is just what we need! Based on the structure description and example, it is not difficult to design the subsystem. The architect can also know the consequences of such a pattern, such as “vary subjects and observers independently”, “reuse subjects without reusing their observers, and vice versa” and so on.

This example seems straightforward and perfect, but in practice, because the system is much larger and more complex, it is not easy to apply these patterns to practical contexts. In fact, this method provides you

a valuable list of the styles and patterns, but does not tell you much about how to use this knowledge base. This makes the architecture design less formalized and more like intuitive craftsmanship than rational engineering.

Much work has been done to extend this methodology and make formal reasoning possible. Here we discuss two promising research, namely goal graph based reasoning and attribute-based architectural styles.

2.1 Goal Graph Based Reasoning

The objective of the approach is to make the reasoning structure behind a pattern explicit, and amenable to systematic analysis [11]. To achieve such goals, this approach use goal graphs to express the effects of the patterns on various requirements.

1. Represent requirements as design goals: Both FRs and NFRs are represented as goals to be achieved. In particular, the NFRs are denoted by NFR softgoals. Here "soft" means they typically do not have clear-cut criteria of achievement.
2. Show the relationships among the goals: The goals, especially the NFR goals, are not independent. Their relationships are explicitly expressed in the goal graph.
3. Show how known solutions achieve goals: The solutions in patterns are represented as operationalizing softgoals. On one hand, the operationalizing softgoals turn those goals into solutions. On the other hand, they are still treated as goals because there are still different ways for achieving them.
4. Identify unintended correlation effects among goals and solutions: The side effects of patterns can also be explicitly specified in graphs using correlation links.
5. Show how alternative solution structures contribute differently to goals: Each proposed solution could be analyzed in terms of its NFR softgoal achievements.

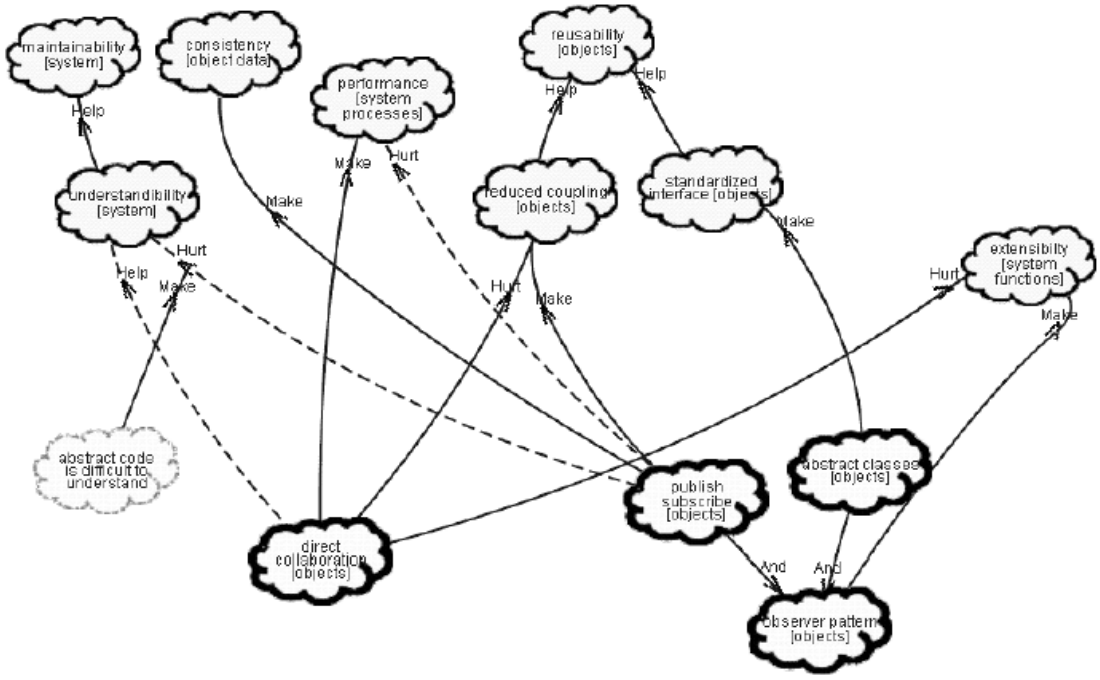

Figure 2. The goal graph for Observer Pattern [11]

Figure 2 shows the goal graph for Observer pattern. In the graph, the NFR softgoals are denoted as the light solid-line clouds, such as maintainability, performance, and reusability. The operationalizing softgoals are denoted as clouds with thick solid borders, such as observer pattern, direct collaboration, publish subscribe, etc. The edges with "Make" mean the positive contributions among softgoals and operationalizing softgoals and edges with "Hurt" mean negative contributions. The "And" refinement into two sub-softgoals means both sub-softgoals are needed to be achieved in order to achieve the parent goal. For example, both publish-subscribe and abstract classes are required in order to implement Observer pattern. The dotted lines with "Help" and "Hurt" specify the positive or negative side effects.

Then we will show how to apply this NFR graph during design with functional structure. We still use the aforementioned example of managing display modes.

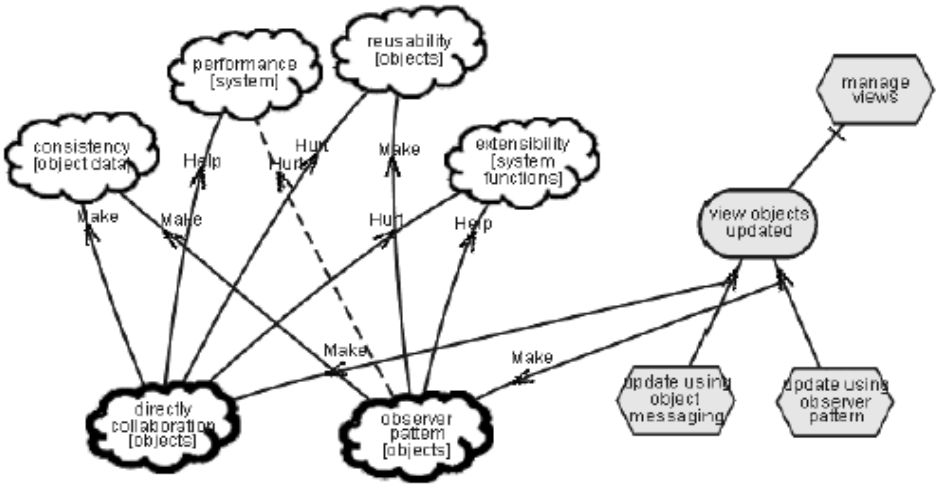

Figure 3. Applying the Observer pattern during design [11]

In Figure 3, the left side is a "collapsed" version of the Observer pattern. On the right side is a fragment of a functional elaboration of the software system under development. The function "manage views" is further refined as "view objects updated" functional goal. A functional goal can be achieved using different ways, e.g. in this case the "view objects updated" goal has two possible solutions as "update using object messaging" and "update using observer pattern". Each solution points to the functional goal through "means-ends" links. The "means-ends" links are related to the operationalizing softgoals through "design justification links". From these links and pattern goal graph, the architect can understand and analyze the contributions of each solution on the goals, either positive or negative.

2.2 Attribute-Based Architectural Style

The second extension is attribute-based architectural style (ABAS), which explicitly associates a reasoning framework (whether qualitative or quantitative) with an architectural style [16][24]. These reasoning frameworks are based on quality attribute-specific models, which exist in the various quality attribute communities. Each ABAS is associated with only one attribute reasoning framework. For those architectural styles that are interesting from different points of views, they may have various ABASs. For example, there may be distinct pipe-and-filter performance and pipe-and-filter reliability ABASs.

An ABAS is defined as a triple [16]:

1. The topology of component types and a description of the pattern of data and control interaction among the components (as in the standard definitions).
2. A quality attribute-specific model that provides a method of reasoning about the behavior of component types that interact in the defined pattern.
3. The reasoning that results from applying the attribute-specific model to the interacting component types.

The first problem addressed here is how to characterize a quality attribute precisely. In the quality attribute-specific model, quality attribute information has three parts: external stimuli, responses and architectural decisions. External stimuli are the events that cause the architecture to respond or change. These measurable/observable quantities are described in the responses section of the attribute characterization. Architectural decisions are aspects of an architecture that have a direct impact on achieving attribute responses. For example, the model of performance is shown in Figure 4, 5, 6 [24].

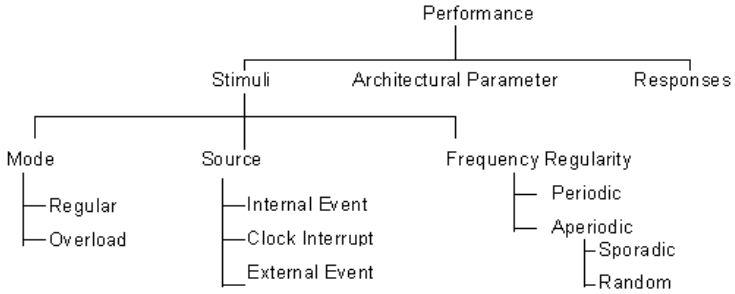

Figure 4. Stimuli of the performance model [24]

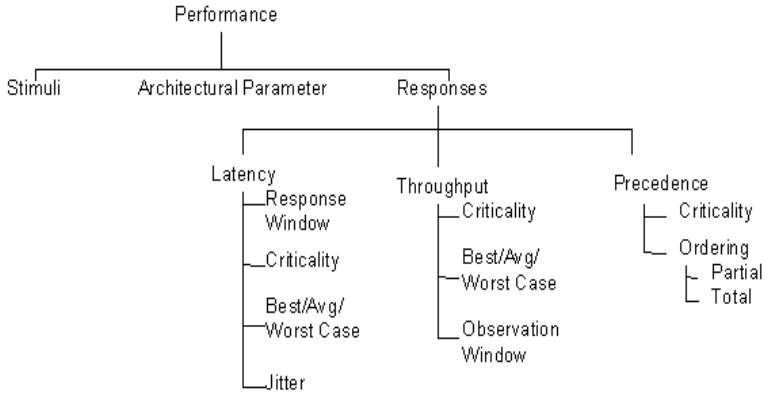

Figure 5. Responses of the performance model [24]

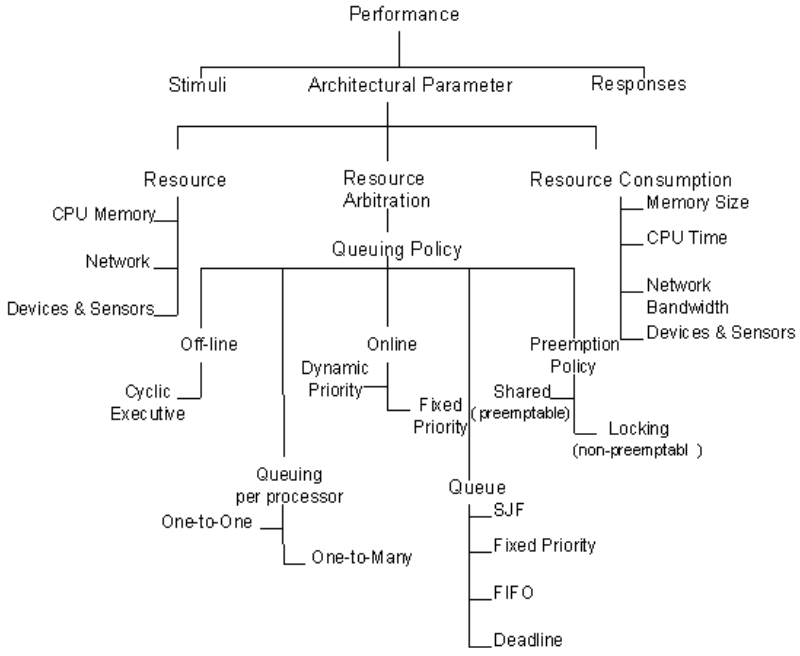

Figure 6. Architectural parameter of the performance model [24]

More information about ABAS, especially how to use them in the architecture design and analysis, can be found in [16][24].

3. Multiple-View Model

One difficulty arising in architecture design is that it is so complex and that different stakeholder may be interested in different aspects of the architecture. How to deal with the complexity? A popular solution is multiple-view model that addresses the different aspects of the system with different views. Various multiple-view models have been developed [3][12][17] and the most well-known model is perhaps the "4+1" view model presented by Rational Software Corporation [17].

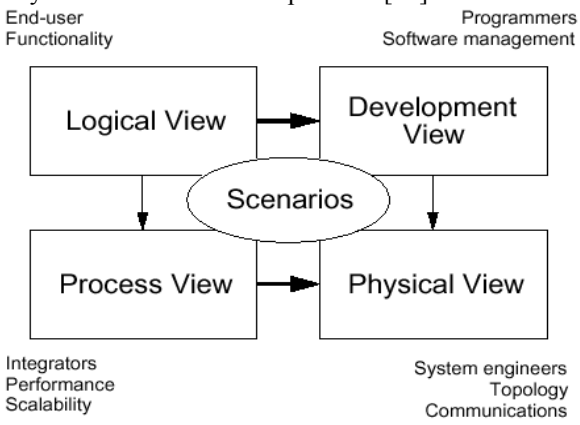

Figure 7. "4+1" view model [17]

Figure 7 shows the "4+1" views in the model. The meanings of these views are explained in the following:

1. The logical view concerns about the functional requirements, i.e. what the system should provide in the terms of services to its users. The logical view is tied closely to the application domain. In this view, the functionality of the system is mapped to architecture elements called conceptual components and the coordination and data exchange are handled by elements called connectors. For examples, the pipes and filters style may have filters as components and pipes as connectors. Since this view addresses the concerns of the end users, it usually uses domain terms and is independent with the software and hardware details.
2. The process view takes into account some non-functional requirements, such as performance, scalability, etc. The execution view defines the runtime entities and their attributes. It also has components and connectors, the components in this view are tasks and the connectors are message, RPC, event broadcast and so on. The users of this view are mainly the system designers and integrators.
3. The development view is of interest to developers and project managers. In this view, the components and connectors are mapped to subsystems or modules. It's focused on the actual software module organization on the software development environment. The software is packaged into subsystems and organized as layers.
4. The physical view takes into account the concerns such as availability, reliability, scalability, etc. The work in this view is mapping various elements, e.g. networks, processes, tasks, onto the various nodes.
5. The "+1" view is scenarios. The scenarios are in some sense the most important requirements. The scenarios serve both as a driver to discover the architectural elements during the architecture design and as a validation and illustration role after design is complete. The scenarios should be specific. So a statement as "the system should be modifiable" is meaningless. Instead, one possible scenario about modifiability may be "it should be easy to add new features of the following type…"

Compared with other methods, multiple-view model may be the most mature and widely used one. Each view has some specific notations or tools to support. For example, you can use class diagrams and class utilities in Rational Rose to draw logical view, Software Architects Lifecycle Environment (SALE) in Universal Network Architecture Service (UNAS) to draw process view, Apex or Rational Rose to draw development view, UNAS to draw physical view and use case diagrams to draw scenarios.

In fact, not all the views have to be developed for an architecture. This method provides a framework and it is up to the architect to decide which views are useful for a project. For example, the process view may be

omitted if there is only one process in the software. In [17], an iterative process is also present as a guideline for the design activities and it is basically a scenario-driven approach.

This model divides a complex architecture into some loose-coupling views from different perspective. This makes it much easier for various users to understand the architecture. It also makes the design job simpler and clearer. Architects and engineers can work separately and concurrently on different views, although the efficient communications and coordination are by all means very important. Another benefit of this model is that it allows the architect to apply the aforementioned design methods to each single view, such as different architectural styles can be applied to different views without disturbance.

One potential drawback of such a model is that there is not a clear boundary of architectural design and detail design. All these views can be used in both designs and those notations, such as class diagrams and use case diagrams, can also be used in detailed design. This makes the output of architecture design not well defined.

Another concern is that UML is a general purpose modeling language, not an ADL. It emerges from object-oriented design and is commonly used in the context of detailed design. Although it can support most architectural concepts, it lacks the semantics needed for extensive analysis. More discussion about the UML in architecture design can be found in [13][18].

4. Evaluation and Transformation Based Design

Similar to the prototype method in the traditional software design, this paradigm advocates for meeting the FRs first and then meeting the NFRs by iterative evaluations and transformations [5].

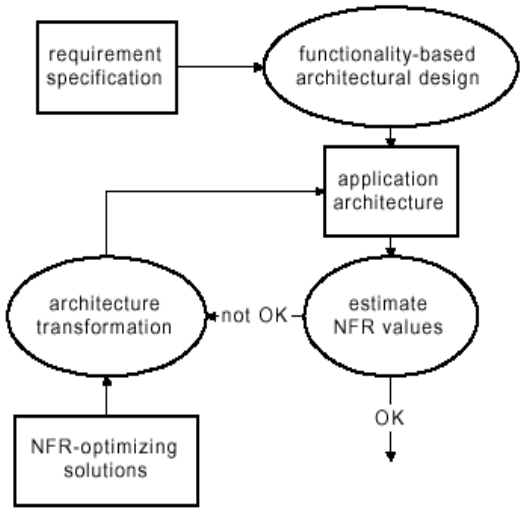

Figure 8. Outline of evaluation and transformation based method [5]

The process starts with the functionality based architectural design. Although the architect may try to make the design meet the NFRs by experience, those NFRs are not explicitly addressed in this stage. Then this design is evaluated using qualitative or quantitative assessment techniques. The estimated NFR values are compared to the requirements. If the estimations are as good or even better than the requirements, the architecture design is finished. Otherwise, some transformation has to be done to optimize the original design in order to meet the NFR. Then the new architecture is assessed again and the process is repeated. To make this methodology feasible, two key problems are addressed. The first problem is how to estimate the NFR values for an architecture design. The second problem is how to transform an existing design and get better NFR values.

4.1 Architecture Evaluation

In [5], four approaches are presented to estimate NFR values, i.e. scenarios, simulation, mathematical modeling and object reasoning.

The scenarios here have the same meaning with the scenarios in multiple-view model. The scenarios-based evaluation is perhaps the most well-investigated and widely used evaluation method [15]. But the effect of this method is heavily dependent on the quality of the scenarios.

Simulation complements the scenario-based approach in that simulation is particularly useful for evaluating operational NFRs, such as performance and fault-tolerance, whereas scenarios are more suited for evaluating development NFRs, such as maintainability and flexibility. The simulation method requires the main components of the architecture being implemented and others are simulated as context.

Mathematical modeling allows for static evaluation of architectural designs. It utilizes some existing models from various research communities, such as high-performance computing, reliable systems, real-time systems, etc.

Objective reasoning method is based on logical arguments. Experienced software engineers and architects often have valuable insights that may prove extremely helpful in architectural designs. This approach is different from the other approaches in that the evaluation process is less explicit and more based on subjective factors as intuition and experience. This kind of analysis often starts with a feeling that something is wrong. Based on that, an objective argumentation is constructed either based on one of the aforementioned approaches or on logical reasoning. For instance, an experienced architect may identify a maintainability problem and, to convince others, define a number of scenarios that illustrate this.

In general, my personal opinion is that the methods of scenarios and objective reasoning are more promising and useful. I don't think either the simulation or the mathematical modeling would be popular in practice. The first reason is that the results are not so meaningful because both the input value and the model are not accurate enough. Te second reason is that in most cases they are more time-consuming and need more techniques.

4.2 Architecture Transformation

Once the NFR values are estimated and do not meet the requirement specification, the architects should analyze the design and decide the causes. Then one may decide to either make changes to the presumed evaluation context (i.e. modify the evaluation) or to make changes to the architecture design. Three methods can be used to transform the architecture [5].

The first way is the use of patterns, i.e. architectural styles or design patterns.

The second type of transformation is the conversion of a NFR into a functional solution. One well-known example is adding the exception handling modules to increase the fault-tolerance value.

The third possible transformation is distributing requirements. This kind of transformation deals with NFRs using the divide-and-conquer principle. One can either divide a NFR to a set of components or divide a NFR into two or more functionality-related NFRs. For example, in a distributed system, fault-tolerance can be divided into fault-tolerant computation and fault-tolerant communication.

5. Architecture-Based Product Lines Design

One of the important trends of architectural design is the desire to exploit commonality across multiple products [9]. A software architecture is an asset that an organization creates at considerable expense. This expense can and should be reused. Architectural based product lines tries to reuse the architecture design in a family of software systems and are proving to be a significant success for many organizations.

When creating a product line, new challenges are encountered that do not occur in single product developments. First, in the product lines approach, one must consider requirements for the family of the systems and the relationships between those requirements and the ones associated with each particular instance. In particular, the architecture for the product line should be easily instantiated or extended for new products. Second, the creation and management of a set of core assents is also challenging. This needs more work on the documentation and formalization.

The development process for the architecture-based product lines method is shown in figure 9 [3].

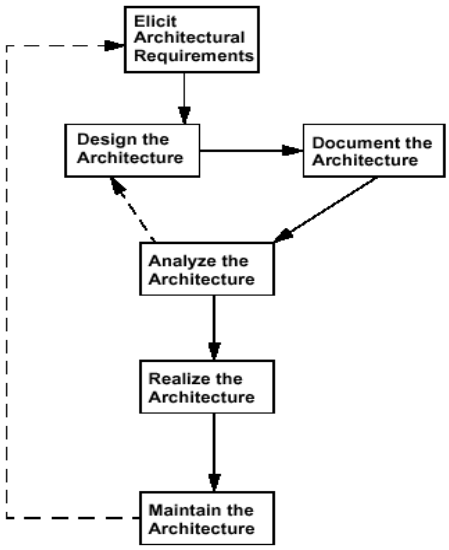

Figure 9. Steps of architecture-based product lines design [3]

Each of these steps includes the definitions of inputs, the constructive activities, the validation activities and the outputs. Compared with other methods, product lines design defines the documentation step explicitly.

Here I only discuss the step of the architecture design. Figure 10 shows the diagram of architecture design.

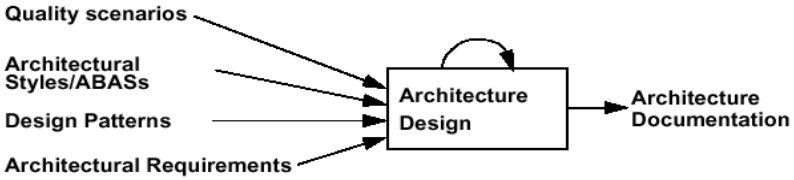

Figure 10. Architecture design in product lines [3]

The input of this stage has four parts: architectural requirements, quality scenarios, architectural styles/ABASs (Attribute Based Architecture Styles as discussed before), and design patterns [3].

The architectural requirement also includes the FRs and NFRs. One distinct point here is that in the architectural requirements, the architecture must be suitable for a product line so that there should exist four or five different architectural variation points that corresponds to different systems in the product line.

The quality scenarios here are for the whole family of the system, not for some specific single system. These scenarios are called abstract scenarios, which make tracing the scenarios much more complex and difficult.

The architectural/ABASs and design patterns serve as knowledge bases supporting the constructive activities.

The design activity in product lines design is based on the aforementioned multiple-view model. It defines the design process more explicitly and further divides it into 4 stages [3].

The first stage starts with a list of architectural requirements and a list of classes of functionality derived from the functional requirements. The goal of the first step is to develop a list of candidate subsystems. All the classes of functionality are automatically candidate subsystems. Other candidate subsystems are derived from the architectural requirements. For example, if the requirement is to allow for the change of operating system, then we may add a virtual operating system adaptor to meet the satisfy the requirement. Some

architectural may have multiple possible choices. The enumeration comes from the architectural style, design pattern and the architect's experience.

The task of the second stage is to choose the subsystems. Each candidate subsystem will be categorized as an actual subsystem, a component in a larger subsystem, or expressible as a pattern to be used by the actual systems. The first two stages are basically carried out in the logical view and development view.

In the third stage, the structure of process view and/or physical view should be taken into account. The architect should decide how to distribute the subsystems across several physical nodes or across different processes.

The quality scenarios are used in the last stage as the primary validation mechanism. The proposed structures are examined to see if the scenarios are achievable at the current level. If not, the design at this level must be reconsidered.

In general the architecture design in product lines is more formal and time-consuming than the other methods. For example, at each step the structure has to be validated and the documents have to be completed. Although all these will obviously increase the cost of architecture design, its practitioners argue that the benefits got from the architecture reuse will outweigh the expenses.

6. Discussions

From the survey over the modern methodologies of software architecture design, some trends can be recognized.

1. The non-functional requirements (NFRs) become the main focus. Different from most traditional software design methodologies, which are focused on the FRs and deal with NFRs intuitively, modern software architecture design regard the NFRs as important as, if not more than, the FRs. One central task of these methods is to make non-functional qualities observable and predictable, for example, in the pattern-based design, a number of structures are presented as guidance of good qualities; in multiple view model, the functional view (logic view) is separated from the other views; and in the evaluation and transformation based design, NFR values are estimated.
2. Architecture styles and design patterns play an important role in all the methods. These patterns serve both as the knowledge base to look for good design paradigms and as an effective way to analyze the non-functional qualities. The latter point becomes more obvious in the goal graph based reasoning and the attribute based architecture styles.
3. The combination of architecture design with traditional software design methodologies seems promising. In the multiple-view model, architecture design is carried out within the object-oriented framework. And in [5][6], the combination of rapid prototyping and architectural focus is shown to be feasible.
4. All the methods are a process of iterative design. This makes evaluations of NFRs more important. Although some quantitative evaluation methods have been presented, they are by all means naïve. In the near future, the most widely used methods are still qualitative, especially scenario-based reasoning.

In the end, I want to address some problems exiting in the modern architecture design.

1. Improve the notations. A number of ADLs are invented in the research communities, but most, if not all, of them are used only within small groups. This makes them hard to become mature. In the industry, UML is widely used. But it is not an architecture language pre se. It is difficult to carry out extensive reasoning and it often mixes the architectural design with the detailed design. Therefore, it has become a serious problem to use different notations in research community and industry. Because the pervasiveness of UML in industry, a promising solution is to extend UML with architecture features. Much work is going on in this direction, e.g. [13][18].
2. Improving the tools. The involvement of NFRs makes the architecture design much more complex and time-consuming than traditional software design. In practice, the development of software design is also tradeoffs among quality, cost and time. Without strong supports from tools, many architects would have to give up architectural design methodologies in order to save time and money. While much progress has been made in this field, e.g. the automated support for architectural styles in AESOP [20],

it is still far from enough. I believe whether software architecture would be pervasive, in some sense, is determined by whether there are excellent tools or not.

3. How to utilize the legacy systems? Most work addressing the architecture design is focused on the green-field. The other important aspect, designing based on preexisting systems, is seldom mentioned. One possible reason may be it is even more difficult. But ignoring such a problem is by no means a good idea, because legacy systems are so common. After all, there are some researches addressing such a problem. For example, in [22], an inductive method for discovering design patterns from preexisting systems is presented.

References:

[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. CMU/SEI-96-TR-025, 1996.

[2] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F.Peruzzi. The Architecture Based Design Method. CMU/SEI-2000-TR-001, 2000.

[3] L. Bass and R. Kazman, Architecture-Based Development, CMU/SEI-99-TR-007, 1999.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. Jon Wiley and Sons, 1996.

[5] J. Bosch and P. Molin. Software Architecture Design: Evaluation and Transformation.

[6] M. Christensen, C. H. Damm, K. M. Hansen, E. Sandvad, and M. Thomsen. Design and Evolution of Software Architecture in Practice. Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages.

[7] P. C. Clements and L. M. Northrop. Software Architecture: An Executive Overview. CMU/SEI-96-TR-003, 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison-Wesley, 1995.

[9] D. Garlan. Software Architecture: a Roadmap, The Future of Software Engineering, ACM Press, pp. 91-101, 2000

[10] D. Garlan and M. Shaw. An Introduction to Software Architecture. IEEE Transaction on Software Engineering 21(4), pp. 269-386. 1993

[11] D. Gross and E. Yu. From Non-Functional Requirments to Design through Patterns. Sixth International Workshop on Requirements Engineering: Foundation for Software Quality, Stockholm, Sweden, June 2000,

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

[13] C. Hofmeister, R. L. Nord, and D. Soni. Describing Software Architecture with UML. Proceedings of the First Working IFIP Conference on Software Architecture, 1999.

[14] R. Kazman. Software Architecture. In Handbook of Software Engineering and Knowledge Engineering, S-K Chang (ed.). World Scientific Publishing, 2001.

[15] R. Kazman, M. Klein, and P. Clements. ATAM: Method for Architecture Evaluation. CMU/SEI-2000-TR-004

[16] M.H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-Based Architecture Styles. Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, 225-243, February 1999.

[17] P. Kruchten, Architectural Blueprints----The "4+1" View Model of Software Architecture. IEEE Software 12 (6), pp. 42-50. November 1995.

[18] N. Medvidovic and D. S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. Proceedings of the First IFIP Working Conference on Software Architecture, San Antonio, TX, February 1999.

[19] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture. Patterns for Concurrent and Networked Objects. Jon Wiley and Sons, 2000.

[20] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[21] S. Shlaer and S. J. Mellor. Recursive Design of an Application-Independent Architecture. IEEE Software. January 1997.

[22] F. Shull, W. L. Melo, and V. R. Basili. An Inductive Method For Discovering Design Patterns From Object-Oriented Software Systems. UMIACS-TR-96-10, University of Maryland, Computer Science Department, 1996

[23] D. Svetinovic and M. Godfrey. Attribute-Based Software Evolution: Patterns and Product Lines Forecasting. ICSE ’02 Buenos Aires, Argentina.
[24] Software Engineering Institute, CMU. http://www.sei.cmu.edu/ata/abas.html.