Quality Attribute Design Primitives and the Attribute Driven Design Method [1]

Len Bass, Mark Klein, and Felix Bachmann

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pa 15213 USA
{ljb,mk,fb}@sei.cmu.edu

Abstract: This paper discusses the understanding of quality attributes and their application to the design of a software architecture. We present an approach to characterizing quality attributes and capturing architectural patterns that are used to achieve these attributes. For each pattern, it is important not only how the pattern achieves a quality attribute goal but also what impact the pattern has on other attributes. We embody this investigation of quality into the Attribute Driven Design Method for designing software architecture.

Introduction

The software architecture community generally believes that quality attributes (such as performance, usability, security, reliability and modifiability) of a software system are primarily achieved through attention to software architecture. This means that the design decisions embodied by a software architecture are strongly influenced by the need to achieve quality attribute goals. We have embarked on an effort to identify and codify architectural patterns that are primitive with respect to the achievement of quality attributes. We call this set of architectural patterns *attribute primitives*. We embody this relationship in a design method for software architecture. In this paper, we provide a status report of this work. In brief, the status is that we have a characterization for six important attributes, we have a list and an organization for attribute primitives to achieve these attributes, and we have modified the Attribute Driven Design method (ADD) to utilize both the attribute characterizations and the attribute primitives. What we have yet to accomplish is to document the attribute primitives in a consistent fashion.

Product lines exist in all domains. Each domain has its own requirements for availability, modifiability, performance, security, or usability. The requirement that there be a product line adds additional complexity to the design task but does not remove the necessity for designing to achieve all of the normal quality attributes for a domain.

This work is a natural extension of the work of various communities:

- The patterns community believes that there are fundamental architectural patterns that underlie the design of most systems.
- Attribute communities have explored the meaning of their particular attribute and come up with standard techniques for achieving their desired attribute.

This work is also similar in spirit and was motivated by the work we have previously reported on Attribute Based Architecture Styles (ABASs) [10]. We believe that architectural styles are compositions of the architectural patterns we are discussing here. By understanding these patterns primitives we believe that we will be better able to understand ABASs and to make the generation of ABASs an easier process.

Codifying a sufficiently rich set of attribute primitives requires a systematic and relatively complete description of the relationship between software architecture and quality attributes. However, this is difficult because:

- Ambiguous definitions. The lack of a precise definition for many quality attributes inhibits the exploration process. Attributes such as reliability, availability, and performance have been studied for

[1] This work supported by the U.S. Department of Defense and Robert Bosch, GmbH

years and have generally accepted definitions. Other attributes, such as modifiability, security, and usability, do not.

- Overlapping definitions. Attributes are not discrete or isolated. For example, availability is an attribute in its own right. However, it is also a subset of security (because denial of service attack could limit availability) and usability (because users require maximum uptime). Is portability a subset of modifiability or an attribute in its own right? Both relationships exist in quality attribute taxonomies.
- Granularity. Attribute analysis does not lend itself to standardization. There are hundreds of patterns at different levels of granularity, with different relationships among them. As a result, it is difficult to decide which situations or patterns to analyze for what quality, much less to categorize and store that information for reuse.
- Attribute specificity. Analysis techniques are specific to a particular attribute. Therefore, it is difficult to understand how the various attribute-specific analyses interact.

Whole books have been written on individual attributes such as performance and reliability. Clearly, it is not our goal to simply reproduce this work. It is our goal to develop some unifying principles that are general to all attributes and codify these principles in a way that empowers architects to perform informed architecture design and analysis. In order to achieve our goal three fundamental questions must be answered:

1. How are quality attributes characterized so that we will know whether the attribute primitives achieve them, and
2. What are the attribute primitives?
3. How are attribute primitives realized in a design process?

We begin the paper by briefing reviewing similar work, and then we discuss the philosophy of our characterization of quality attributes and the particular set of architectural patterns that we choose. We close by presenting the current state of the ADD method as a method for realizing quality attributes within software architectures.

Related work

The first piece of related work is the body of work reported in the most depth in Chung [7]. Chung et al. use “softgoals” as the basis of their use of quality attributes. A softgoal is a goal with no clear-cut definition and/or criteria as to whether it is satisfied or not. They characterize quality attribute goals as softgoals and develop a process that involves viewing a design as a point on a quality attribute space. Each design decision, then, is made on the basis of how it might move the design in the quality attribute space. A design decision might move the design in the positive direction on the modifiability axis and the negative direction on the performance axis. This work has the same goals as ours – the creation of a design with desired quality attributes – but differs in two fundamental respects. First we do not believe that quality attribute goals are softgoals. They can be precisely specified and criteria developed to determine whether they have been satisfied. We describe how we do this in the next section. Secondly, given a set of techniques for achieving a particular attribute, they visualize the design process as simultaneously attempting to satisfy the collection of softgoals for a system. We believe that there is a systematic method of prioritizing and dividing the quality attribute goals that greatly informs the design process. We will see our approach in our discussion of architectural drivers.

The second related work is [6]. Bosch presents a design method that elevates quality attributes from being almost totally ignored to an important player in the design process. This method, however, still places functional requirements as primary and quality requirements as secondary. The method begins with a design that achieves the functional requirements and this design is iteratively transformed into one that achieves quality requirements. The ADD Method assumes that quality attributes drive the design process. Functionality is important in the design process but we believe that the “shape” of an architecture is determined by its quality requirements and, hence, these are the requirements that determine the initial iteration of a design.

Quality Attributes and General Scenarios

Attribute communities have a natural tendency to expand. This is not surprising because many of the attributes are intertwined. However, we are not trying to define an attribute by deciding what is within its purview. For example, we do not care whether portability is an aspect of modifiability or whether reliability is an aspect of security.

Rather, we want to articulate what it means to achieve an attribute by identifying the yardsticks by which it is measured or observed. To do this, we introduce the concept of a “general scenario.” Each 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

For example, a performance general scenario is: “External events arrive at a system periodically during normal operation. On the average, the system must respond to the message within a specified time interval.” This general scenario describes “external events arriving” (stimuli and their source) during normal operation (the context) and being serviced by the system (the system element) with some “latency” (response and response measure). General scenarios for modifiability focus on “changes arriving” and the “propagation of the change through the system.” Security general scenarios combine “threats” with a response of “repulsion, detection, or recovery.” Usability general scenarios tie “the user” together with a response of “the system performs some action.” We have developed a collection of general scenarios for the attributes of availability, modifiability, performance, security, testability, and usability. We have also tested the general scenarios against those developed “in the wild” during architecture evaluations [4] and are reasonably confident that this list will suffice for most applications. Furthermore, our collection of general scenarios is expandable and if we have omitted some, the process does not change.

Also, we are primarily interested in the total collection of general scenarios. Thus, we do not care if multiple attribute communities claim the same general scenario. The use of general scenarios addresses the first two problems that we claimed inhibited codification of the relationship between architecture and quality attributes; that is, attribute definitions and their relationships. [2]

General scenarios can apply to any system. For example, we can discuss a change to system platform without any knowledge of the system, because every system has a platform. Or we can discuss a change in the representation of data (in a producer/consumer relationship) without knowing anything about the type of data or the functions carried out by the producer. This is not the same as saying that every general scenario applies to every system.

For example, “the user desires to cancel an operation” is one usability general scenario stimulus. This is not the case for every system. For some systems, cancellation is not appropriate. Thus, when applying general scenarios to a particular system, the first step is filtering the universe of general scenarios to determine those of relevance.

The second step when applying general scenarios to a particular system is to make the scenario system specific. Since the scenarios are system independent, they are abstractions that must be mapped to particular situations for specific systems. This mapping is not always an easy task. For example, one general scenario is that requests for changes in functionality arrive. Determining which changes are likely for a particular system is often complicated. However, if modifiability is an important quality attribute for the system this challenge must be faced.

In summary, general scenarios provide two key benefits to the architect:

- Encourage and inspire precise articulation of quality attribute requirements
- Provide a checklist to help ensure completeness of quality attribute requirements

[2] Note that even though we advocate defining attributes through general scenarios, attribute names convey meaning and we use them, as appropriate, to define useful categories.

Attribute primitives

An attribute primitive is a collection of components and connectors that 1) collaborate to achieve some quality attribute goal (expressed as general scenario) and 2) is minimal with respect to the achievement of those goals. This is a restriction of the concept of an architectural mechanism introduced by Booch [5].

The primitives that we seek comprise a set of components and their relationships. These components are bound together into an attribute primitive by performing work in synergy to achieve some specific aspect of a quality attribute. Furthermore, the primitives are minimal. If any one of the elements of the primitive is removed then an argument can no longer be made that the attribute primitive achieves the quality attribute goal.

Examples of attribute primitives are a data router, a cache and the components that access it, and fixed priority scheduling. These attribute primitives help achieve specific quality attribute goals as defined by the general scenarios.

- 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 affects or contributes to modifiability.
- Caching reduces response time by providing a copy of the data close to the function that needs it. This contributes to performance.
- Fixed priority scheduling interleaves multiple tasks to control response time. This contributes to performance.

Each attribute primitive is targeted at one or more quality attributes. However, each attribute primitive has an impact on quality attributes that it does not target. Thus, a data router consumes resources (performance), may fail (availability), may introduce vulnerabilities (security) or may impact what information is available to the user (usability). The understanding of the impact of a attribute primitive on quality attributes then has two portions: how does it support the achievement of the quality attribute (more precisely, the general scenario) at which it is targeted and what is its impact on the other quality attributes. We call the impact on other quality attributes *side effects*.

We can also extend the list of attribute primitives and general scenarios to address a wide range of situations. If we introduce a new general scenario, we may need to introduce new attribute primitives that contribute to it. Alternatively, we may discover new attribute primitives for dealing with existing general scenarios. In either case, we can describe the new attribute primitives(s) without affecting existing primitives.

We believe that attribute primitives are the design primitives for achieving system quality attribute behavior. The goal of our work is to identify the specific primitives that elicit quality attributes, and then analyze those primitives from the point of view of multiple quality attributes.

Attribute Driven Design

The Attribute Driven Design (ADD) method is an approach to defining a software architecture by basing the design process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage in the decomposition, 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.

There are a number of different design processes that could be created using the general scenarios and attribute primitives. Each process assumes different things about how to “chunk” the design work and about the essence of the design process. We now discuss ADD in some detail to illustrate how we are applying the general scenarios and attribute primitives and, hence, how we are “chunking” the work and what we believe is the essence of the design process. We begin by discussing where ADD fits into the life cycle.

ADD in life cycle

ADD takes requirements, quality as well as functional, as input. Therefore, ADD's place in the life cycle is after the requirement analysis phase, although, as we shall see, this does not mean that the requirements process must be completed prior to beginning ADD. The output of ADD is a conceptual architecture [9] Thus, ADD does not complete the architecture design but provides the basis for its completion.

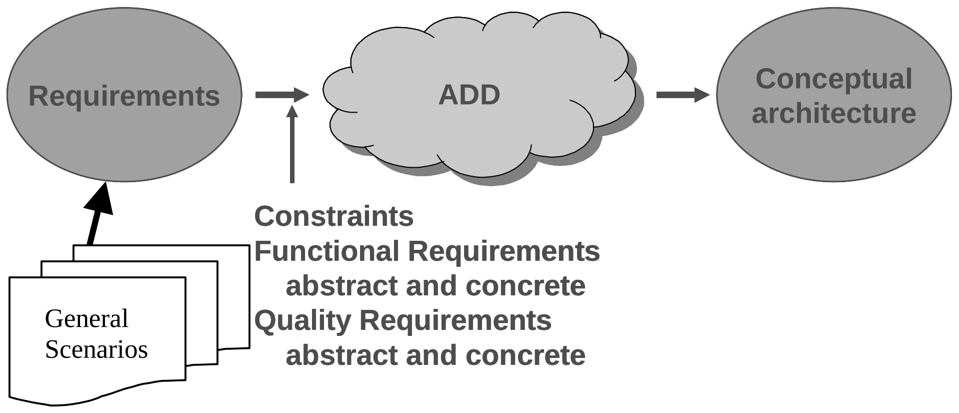

Figure 1 ADD in life cycle

We now briefly discuss both the input and the output of ADD. We also discuss when during the requirement process ADD can begin.

ADD input. The input to ADD is a set of requirements. Any design process has as its input a set of requirements. In ADD, we differ from other design methods in our treatment of quality requirements. ADD requires that the quality requirements be expressed as a set of system specific quality scenarios. The set of general scenarios discussed above act as input to the requirements process and provide a checklist to be used in developing the system specific scenarios.

ADD output. Applying ADD to a system results in a conceptual architecture. A conceptual architecture is a representation of the high-level design choices. It describes a system as containers for functionality and the interactions among them. A conceptual architecture is the first articulation of architecture during the design process and therefore necessarily coarse grained. Nevertheless, it is critical for achieving the desired qualities and it provides a framework for achieving the functionality.

So, if there is a conceptual architecture, is there also a concrete architecture? The answer is yes, of course. The difference is that in a concrete architecture for a specific product, additional design decisions need to be made. This could be, for example, the decision to use the object-oriented paradigm. The conceptual architecture intentionally may have deferred this decision in order to be more flexible. For example, a conceptual architecture usually describes two components as exchanging specific information. It neither decides on the data type for this information nor on the method of exchanging information. To produce an object-oriented architecture, those decisions need to be made. Therefore

component A exchanges information X with component B

from the conceptual architecture becomes the more concrete

component B invokes method E of component A in order to receive an object Y of type X

in the concrete architecture.

As shown in this example, the conceptual architecture provides a framework for the subsequent work. It serves as a blueprint for the concrete architecture. It also provides an organizational framework. Having the conceptual architecture enables project management to organize work assignments, configuration management to setup the development infrastructure and the product builders to decide on the test strategy to just name few. These are all good reasons for having a conceptual architecture as early as possible!

Beginning ADD. Of the requirements to a system, only a few are architecturally significant. We call these architectural drivers. ADD can start as soon as all the architectural drivers are understood. Of course, during the design this assessment may change either as a result of better understanding of the requirements

or as a result of changing requirements. Still, the process can begin when the architectural drivers requirements are known with some assurance.
In the following section we discuss ADD itself.

Steps of ADD

We begin by presenting the steps performed when designing a conceptual architecture using the ADD method. We then discuss the steps in more detail.

1. Choose the design element to decompose. The design element to start with is usually the whole system. All required inputs for this design element should be available (constraints, functional requirements, quality requirements)
2. Refine the element according to these steps:
    A. Choose the architectural drivers from the set of quality scenarios and functional requirements. This step determines what is important for this decomposition.
    B. Choose the attribute primitives and children design element types to satisfy the architectural drivers. This step is designed to satisfy the quality requirements.
    C. Instantiate design elements and allocate functionality from use cases using multiple views. This step is designed to satisfy the functional requirements.
    D. Verify and refine use cases and quality scenarios and make them constraints for the children design elements. This step verifies that nothing important was forgotten and prepares the children design elements for further decomposition or implementation.
3. Repeat the steps above for every design element that needs further decomposition.

Choose Design Element
We use the term "design element" to describe either the portion of the system being decomposed or the elements of the decomposition. The use of a single term allows us to express the recursiveness of the method. The following are all design elements: system, "conceptual subsystem", "conceptual component". The decomposition typically starts with the top most design element, "the system" and is then decomposed into "conceptual subsystems" and those get further decomposed into "conceptual components". We have used ADD to design the conceptual architecture for a fairly large information system as well as for small embedded systems. So far two levels of decomposition have been sufficient. Nevertheless, there is nothing within the method that would prevent further decompositions.
The decomposition results in a tree of parents and children. The order of decomposition depends on many factors and is not predefined by ADD. Some of the factors are

- *The personnel on the team.* People with specific expertise may be engaged to explore a particular portion of the tree in some depth.
- *The incorporation of new technology.* If new technology is to be used such as middleware or operating system, then prototypes will need to be constructed. This will both aid understanding of the capabilities and limitations of the new technology, and give the architecture team expertise in using the technology. This corresponds to traversing the tree to some depth in one area.
- *The knowledge of the domain.* If the architecture team has extensive knowledge of the domain then a breadth first traversal of the tree is the likely pattern since no exploration need be done.

For the very first step the design element chosen is usually the system that has to be developed. To apply ADD to the selected design element the required inputs, which are the constraints, the functional and quality requirement have to be available.

Choose the architectural drivers
Some requirements are more influential than others on the design of the architecture and the decomposition of each design element. Influential requirements can be

- Functional – e.g. the system has to support administrative tasks
- Quality – e.g. during landing and take-off the landing gear has to function with no downtime.
- Business – e.g. build a product line

Business requirements will translate into both quality and functional requirements. The business goal to create a product line can be translated into modifiability necessary to build the envisioned products. Time to market is another business goal but this could be translated into high reuse of existing products. Business goals can also be functional such as the product has to have a specific features because this protects a market niche.

Architectural drivers are the combination of functional and quality requirements that "shape" the architecture or the particular design element under consideration. The quality architectural drivers will be found among the top priority quality requirements for the design element.

Functional architectural drivers are determined by looking at the functional requirements the design element has to fulfill. The functionality usually will be grouped according to some criteria and those groups influence the decomposition. We will talk about this in more detail in the step were design elements are instantiated.

Determination of architectural drivers is not always a top-down process. Sometimes detailed investigation is required to understand ramifications of particular requirements. For example, to determine if performance is an issue for a particular system configuration a prototypical implementation of a piece of the system may be required.

As a rule of thumb, there should be a small number of quality architectural drivers because they determine the basic structure and are usually in conflict with each other. The purpose of the prioritization of quality requirements is to enable a principled choice of the architectural drivers. The number of functional architectural drivers is not an issue. They are used to determine the number of instances within the structure.

In essence, the style of the architecture is determined by the quality architectural drivers and the instances of the element types defined by that style are determined by the functional architectural drivers. For example, we may have chosen the attribute primitive "data-router" to support modifiability. The data-router 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 a data value, and a guidance as well as a 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. The "data-router" element type might be instantiated into a blackboard element.

Note that we are basing our decomposition of an element on the architectural drivers. Other requirements, both functional and quality apply to that design element but by choosing the architectural drivers, we are reducing the problem to satisfying the most important requirements, not necessarily all of the requirements. The less important requirements are satisfied within the constraints of the most important requirements. This is an important difference between ADD and other design methods.

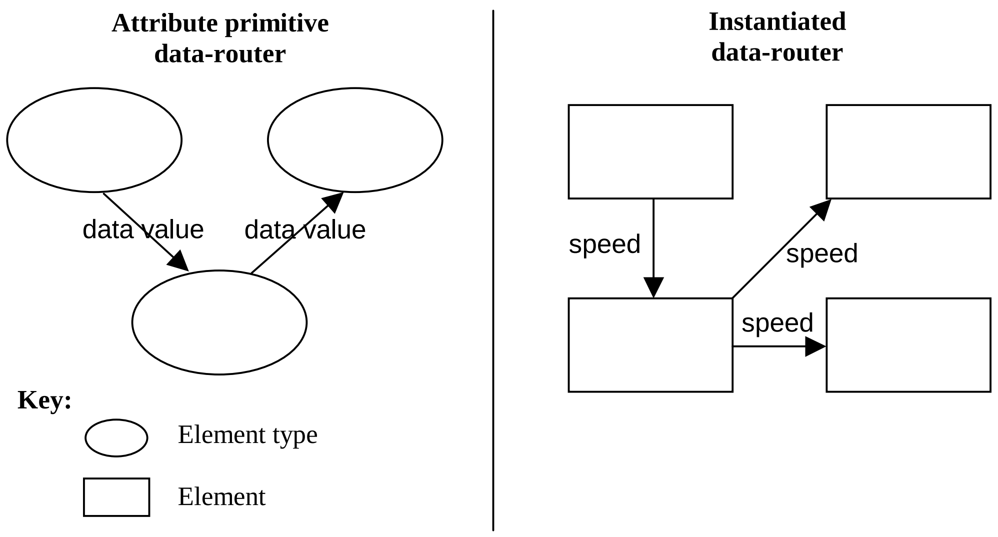

Figure 2 Instantiation of an attribute primitive

Choose the attribute primitives

As discussed earlier, for each quality there are identifiable attribute primitives that can be used in an architecture design to achieve a specific quality. Each attribute primitive is designed to realize one or more quality attributes but has an impact on other quality attributes. In an architecture design a composition of many such primitives is used to achieve a balance between the required multiple qualities. Detailed analysis of the achievement of the quality and functional requirements is done during the refinement step.

Table 1 shows some of the commonly used attribute primitives. It is not our intention to present an exhaustive table here. This table should give an idea about possible primitives for some of the important quality attributes. Every attribute primitive write-up should have a more detailed description that explains how the mechanism works to achieve its quality attribute, when to use it, and what the side effects are. This is ongoing work at the Software Engineering Institute. Examples of attribute primitive descriptions can be found in [3].

The goal of this step is to establish an architectural style that satisfies the quality architectural drivers by composing selected attribute primitives.

Two main factors guide the selection of attribute primitives. The first factor is the drivers themselves and the second factor is the side effects an attribute primitive has on other qualities. As we said before, our vision for this step is that there is a list of attribute primitives with description and analysis for each primitive including an understanding of how to analyze for size effects.

Table 1: Sample attribute primitives

Performance	Modifiability	Security	Availability	Testability	Usability
Load balancing	Data Router	Encryption	Ping/Echo	Monitors	Separation of command from data
Priority assignment	Data repository	Integrity	Voting	Backdoor	Separation of data from the view of that data
Fixed priority scheduling	Virtual machine	Firewalls	Recovery blocks	open APIs	Replication of commands
Cyclic Executive	Interpreter	Mirroring of databases	Atomic transactions		Recording
Client-Server		Audit trail	Checkpoints		Explicit models for Task, User, System

For example, one primitive to achieve modifiability is an interpreter. An interpreter makes easier the creation of new functions or modification of existing function. Macro recording and execution is an example of an interpreter. This is an excellent mechanism for achieving modifiability at run-time, but it also has a strong negative influence on performance. The decision to use an interpreter now depends on the importance of performance. A decision may be made to use an interpreter for a portion of the style and other attribute primitives for other portions. In any case, having available both a description of the interpreter and an understanding of its impact on performance would certainly assist the architect in, first, just thinking of an interpreter as an option, and secondly, in evaluating its utility in a particular context.

For another example, consider the situation where both modifiability and performance are architectural drivers; a common problem. Usually those two qualities hinder each other. One attribute primitive for modifiability is virtual machine and one attribute primitive for performance is cyclic executive scheduling.

A virtual machine introduces data transformations and additional checks at the interface of the virtual machine; an obstacle for achieving performance. A cyclic executive on the other hand delivers real-time performance because the scheduling is implemented into the code. The side effect is that this introduces time dependencies into the code, e.g. functions have to run in a specific sequence and the correctness of a result of the function may depend on the execution time of the functions before. This is a barrier for achieving modifiability. Again, having an enumeration of the possible mechanisms as solutions and the side effects of those solutions enables the architect to determine the tradeoffs implicit in a proposed solution.

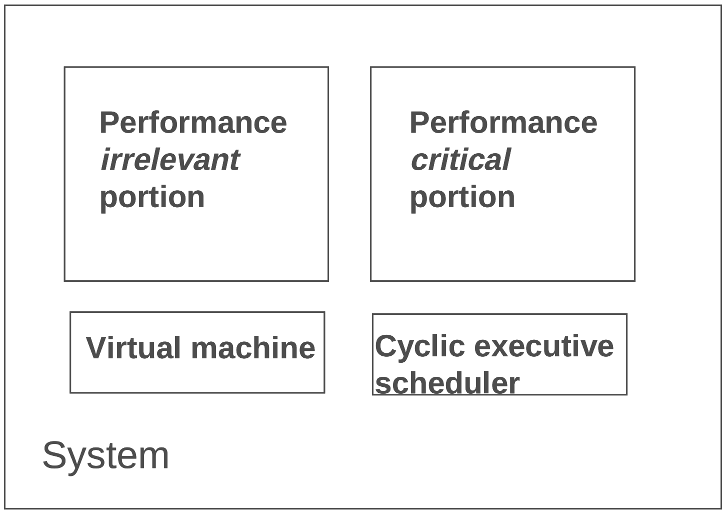
Performance *irrelevant* portion
Performance *critical* portion
Virtual machine
Cyclic executive scheduler
System

Figure 3: Trade-off between performance and modifiability

Clearly, we cannot use both attribute primitives without any restrictions because they very likely would neutralize each other. The next step would be to determine if the mechanisms could be used at least for parts of the system in a beneficial way. In many systems the answer is yes. A system can be divided into a performance critical part and the rest. The cyclic executive scheduler would be used only for the critical part and the virtual machine for the uncritical part. Figure 3 shows the resulting structure.

This example shows that a design of at least parts of a conceptual architecture is feasible without considering any functionality.

Instantiate design elements and allocate functionality using multiple views

In the section above we showed how the quality architectural drivers determined the decomposition structure of a design element. As a matter of fact, in that step we defined the *type* of the elements of the decomposition step. We now show how those design element types will be instantiated by using the functional architectural drivers.

In the example shown in Figure 3, we defined a performance irrelevant part running on top of a virtual machine. The virtual machine usually is an operating system or perhaps communication middleware. The software running on top of the virtual machine is typically an application. Mostly in a concrete system we will have more than one application; one application for each "group" of functionality. The functional architecture drivers help to determining those groups, or instances of the specific element type.

The functional drivers are derived from the abstract functional requirements (e.g. features) or concrete functional requirements (e.g. use cases, list of responsibilities, etc.). Some criteria that can be used to group functionality are:

1. Functional coherence. Requirements grouped together should exhibit low coupling and high cohesion. This is a standard technique for decomposing function to support modifiability.
2. Similar patterns of data or computation behavior. Those responsibilities that exhibit similar patterns of data and computation behavior should be grouped together, e.g. accessing a database in a similar fashion. This will support future performance considerations.
3. Similar levels of abstraction. Responsibilities that are close to the hardware should not be grouped with those that are more abstract. This will support modifiability.
4. Locality of responsibility. Those responsibilities that provide services to other services should not be grouped with purely local responsibilities. This will support reusability of services.

Applying this to the example from Figure 3, we may find that we have two different performance critical parts, such as reading and computing sensor input and keeping a radar display current. We may also find that on the performance irrelevant side there should be several separate applications like diagnosis, administration, help system, etc. Therefore, applying the functional architectural drivers would result in the decomposition like the one shown in Figure 4.

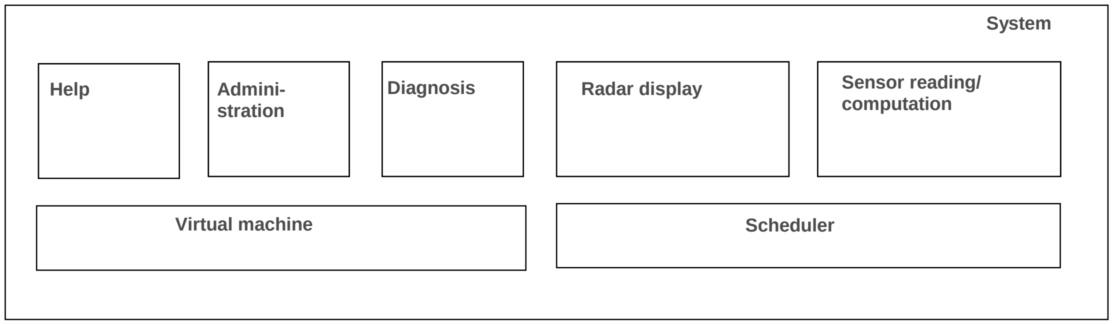

Figure 4: Applying functional architectural drivers

The result of this step is a plausible decomposition of a design element. The next steps verify how well the decomposition achieves the required functionality.

A. The first step is assigning the functional requirements of the parent element to its children by defining *responsibilities* of the children elements. The functionality represented by the functional architecture drivers should be obvious. Other functionality will not be this apparent. Introducing a virtual machine is a nice concept, but what exactly will the functionality be?

   Applying use cases that pertain to the parent element help to gain more detailed understanding about the distribution of functionality. This also may lead to adding or removing children elements in order to fulfill all the required functionality. At the end, every use case of the parent element must be representable by a sequence of responsibilities of the children elements.

B. Assigning responsibilities to the children in a decomposition also leads to discovery of necessary *information* exchange. This creates a producer/consumer relationship between those elements, which needs to be recorded. At this point in the design it is not important to define how the information is exchanged. Is the information pushed or pulled, is it a message or a call parameter; all questions that need to be answered later in a concrete architecture and/or component design. At this point only the information itself and the producer and consumer are of interest.

C. With the usage of attribute primitives came specific patterns on interactions between the element types of the primitive. Those interaction patterns can mean things like “calls”, “subscribes to”, “notifies”, etc. The attribute primitives that are selected define the correct meaning. For example, the applications on top of the virtual machine in Figure 4 have a “calls” relation to the virtual machine. This was introduced by the attribute primitive virtual machine to minimize dependencies. A “schedules” relation exists between the scheduler and the two elements of top of it. This interaction pattern was introduced by using the attribute primitive “Cyclic Executive Scheduler”.

These steps should be sufficient to gain confidence that the system can deliver the desired functionality. In order to check if the required qualities can be met, we usually need more than just the structural information discussed so far. Dynamic and run-time information is also required to analyze the achievement of qualities like performance, security, reliability, to name just few. Therefore, we introduce some architectural views that help focus on different aspects of the conceptual architecture.

Representing the architecture with views

A software architecture is a complex entity that cannot be described in a simple one-dimensional fashion. We use the concept of views to give us the most fundamental principle of architecture documentation: Documenting an architecture is a matter of documenting the relevant views.

What are the relevant views? It depends on your goals. Architecture documentation can serve many purposes: a mission statement for implementers, a basis for analysis, the specification for automatic code generation, the starting point for system understanding and asset recovery, or the blueprint for project planning. For ADD we suggest using the following views:

~ The *module view* shows the structural elements and their relations. We also use this view to document responsibilities of the elements.
~ The *concurrency view* shows the concurrency in the system.
~ The *deployment view* shows the deployment of functionality onto the execution hardware.

The number of possible views of an architecture is limitless. So far, we found that on the conceptual level those three views describe the architecture in sufficient detail. If there is a need for a specific system to show other aspects such as run-time objects or component architecture additional views can be introduced.

Module view

In a module view a system's software is decomposed into manageable units and therefore is one of the important forms of system structure. It determines how a system's functionality is partitioned into separable parts, what kinds of assumptions each part can make about services provided by other parts.

When designing a conceptual architecture using ADD, modules of the module view are used to represent the design elements we mentioned before, the system, the conceptual subsystems, and the conceptual components. The modules serve as containers for functionality.

Concurrency view

In the concurrency view dynamic aspects of a system such as parallel activities and synchronization can be modeled. This modeling helps to identify resource contention problems, possible deadlock situations, data consistency issues, etc. Modeling the concurrency in a system most likely leads to discovery of new responsibilities of the design elements, which are recorded in the module view. It also can lead to discovery of new elements, such as a resource manager, in order to solve issues like concurrent access to a scarce resource.

The elements of a concurrency view are "virtual threads", which describe an execution path through the system or parts of it. This should not be confused with operating system threads (or processes), which implies other properties like memory/processor allocation. Those properties are not of interest on the level of the conceptual architecture. Nevertheless, after the decisions for an operating system and deployment to processing units are made, virtual threads have to be mapped onto operating system threads. This is done during design of the concrete architecture.

The relations between elements in a concurrency view are "control-oriented" relations, such as "synchronizes with", "starts", "cancels", etc and "data-oriented" relations, such as "communicates with".

A concurrency view usually will show the elements of the module view to support understanding of the mapping between those two views. It is important to know that a synchronization point is located in a specific design element so that this responsibility can be assigned at the right place.

To understand the concurrency in a system we found exercising of the following use cases very illuminating:

- Having two users doing similar things at the same time. This helps to recognize resource contention or data integrity problems.
- One user performing multiple activities simultaneously. This helps to uncover data exchange and activity control problems.
- Starting-up the system. This gives a good overview about permanent running activities in the system and how to initialize them. It also helps deciding on an initialization strategy, such as everything in parallel or everything in sequence or any other model.
- Shutting-down a system. This helps to uncover issues of cleaning up, such as achieving and saving a consistent system state.

Concurrency might also be a point of variation. For some products a sequential initialization will work well, while for others everything should be done in parallel. If the decomposition would not support this so far (e.g. by exchanging a component) then this can be an indication that the decomposition needs to be adjusted.

Deployment view

If multiple processors or specialized hardware is used in a system, then issues arise from deploying design elements to the hardware. Using a deployment view helps to determine and design a deployment that supports achieving the desired qualities. It helps to decide if multiple instances of some design elements are needed. For example, a reliability requirement may force us to duplicate critical functionality on different processors. A deployment view also supports reasoning about using special purpose hardware. The elements of a deployment view are "units of deployment". Those units of deployment are aggregates of instances of the design elements. This definition includes two important relations to the design elements, which are "is an instance of" and "aggregation". For every design element the decision need to be made if one or more instances of this element will run on the system. The instances are then aggregated together if they always will be deployed together.

The definition of units of deployment is not an arbitrary procedure. As with the module and concurrency views, the selected quality and functional architecture drivers help determine the units of deployment. Attribute primitives such as replication offers a means to achieve performance or reliability by deploying replicas on different processors. Other primitives such as a real-time scheduling mechanism prohibit deployment on different processors. Functional considerations usually guide the deployment of the parts that are not pre-determined by the selected attribute primitives.

Nevertheless, there is some degree of freedom in specifying the deployment. There might be even variations in the deployment if the architecture has to support multiple products. A product for the low-cost market very likely has a different deployment then a high-end product. Defining the possible deployments might lead to difficulties. Some configurations may not be possible without changing the structure defined so far. Then the assignment of functionality to element types needs to be adapted.

Units of deployments also can be deployed in several instances. For example, it would make sense to aggregate the diagnosis application with the virtual machine and deploy them always together on every machine that runs applications requiring the virtual machine.

The deployment view also shows the "is assigned to" relation that shows how the units of deployment are assigned to the different hardware elements.

The "information" and "interaction" relations from the module view, as well as the crossing of a virtual thread from one element to another require special attention in the deployment view. If deployed on different processors, those relations indicate a communication requirement between the units of deployment. Some design element must have the responsibility for managing the communication and this responsibility must be recorded in the module view.

Define interfaces of the children design elements

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.

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. Those aspects are:

- The module view documents
  - producer/consumer of information
  - certain pattern of interaction, which require elements to provide services and to use services
- The concurrency view documents
  - Interactions among threads, which usually lead to the interface of an element providing or using a service
  - The information that an element is active, e.g. has its own thread running
  - The information that an element synchronizes, sequentializes, and perhaps blocks calls
- The deployment view documents
  - The hardware requirements, such as special purpose hardware
  - Some timing requirements, such as the computation speed of a processor has to at least 10 MIPS
  - Communication requirements, such as an information should not be updated more then once every second.

All those information should be available in the interface documentation of every design element.

Validate and refine use cases and quality scenarios as constraints to children design elements

The steps enumerated thus far amount to a proposal for a decomposition of the design element. This decomposition must be verified and the children design elements prepared for their own decomposition.

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. Currently, this validation is done using informal reasoning. A portion of our vision of the use of attribute primitives is that the verification can be done based on models for achievement of the various attributes. Thus, one of the aspects of ongoing work at the SEI is to deepen our understanding of the models underlying the various attributes and how the primitives achieve them.

Once the decomposition has been verified, the constraints and requirements must be themselves decomposed so that they apply to the children design elements. We began with a list of constraints, quality scenarios and functional requirements such as use cases as input into the decomposition step. We now discuss how each of these is applied to the children design elements.

Functional requirements

The functional requirements should be satisfied by the manner in which the responsibilities of the children elements were determined. Each children design element has responsibilities that derive partially from consideration of a decomposition of the functional requirements. Those responsibilities can be translated into use case for the design element. Another way of defining use cases is to split and refine the parent use cases. For example, a use case that initializes the whole system is broken into the initializations of subsystems. This approach has continuity because somebody can follow the refinement of the use cases.

Constraints

Constraints of the parent element can be satisfied in one of the following ways:

- The decomposition satisfies the constraint. For example, the constraint of using a certain operating system can be satisfied to defining the operating system as a child element. The constraint can be checked as satisfied.
- The constraint will be satisfied by a single child element. For example, the constraint for using a special protocol can be satisfied by defining an encapsulation child element for the protocol. The constraint is moved to this child element.
- The constraint will be satisfied by multiple children elements. For example using the web requires two elements (client and server) to implement the necessary protocols. The constraint is divided and its parts are assigned to the children elements.

Quality scenarios

Quality scenarios also have to be refined and assigned to the children elements.

- A quality scenario may be completely satisfied by the decomposition without any additional impact. Then this scenario can be marked as satisfied.
- A quality scenario may be satisfied by the current decomposition with constraints on children design elements. For example, using layers might satisfy a specific modifiability scenario, which, in turn, will constrain the usage pattern of the children.
- The decomposition may be neutral with respect to a quality scenario. For example, a usability scenario pertains to portions of the user interface that is not yet a portion of the decomposition. This scenario should be assigned to the child element that makes the most sense.
- A quality scenario may not be satisfiable with the current decomposition. If it is an important scenario then the decomposition should be reconsidered. Otherwise the rationale why the decomposition does not support this scenario must be recorded. This is usually the result of a trade-off with other perhaps higher priority scenarios.

At the end of this step we have a decomposition of a design element into its children, where each child element has a collection of responsibilities, a set of use cases, an interface, quality scenarios, and a collection of constraints. This is sufficient to start the next iteration of decomposition.

Conclusion

We have presented our vision of a characterization of quality attributes, a characterization of attribute primitives to achieve qualities and a design method that is based on these characterizations. Our basic premise is that architectural design is driven by quality requirements.

As we have said, this is work in progress. Our current status is that we have a list of general scenarios for six attributes, we have a provisional list of attribute primitives and we have the ADD method. We have applied ADD to several different industrial systems with good success. Our list of general scenarios has been validated through comparison with scenarios generated independently. We have documented several attribute primitives and are currently engaged in documenting additional ones.

References

[1] Bachmann, F.; Bass, L.; Chastek, G.; Donohoe, P.& Peruzzi, F. *The Architecture Based Design Method*. CMU/SEI-2000-TR-001 ADA375851. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000.

[2] Bass, L.; Clements, P. & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison Wesley Longman, 1998.

[3] Bass, L., Klein, M., Bachmann, F. *Quality Attribute Design Primitives* CMU/SEI-2000-TN-017

[4] Bass, L, Klein, M., Moreno, G., *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method,* CMU/SEI-2001-TR-014

[5] Booch, G. *Object Solutions: Managing the Object-Oriented Project*. Reading, MA: Addison Wesley Longman, 1996.

[6] Bosch, J. *Design & Use of Software Architectures,* Addison Wesley, 2000.

[7] Chung, L., Nixon, B., Yu, E. and Mylopoulos, J. *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers, Boston, Ma.

[8] Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. *Design Patterns*. Reading, MA: Addison Wesley Longman, 1995.

[9] Hofmeister, C., Nord, R., Soni, D. *Applied Software Architecture,* Addison Wesley, 2000.

[10] Klein, M.; Kazman, R., Bass, L,; Carriere S.J.; Barbacci, M. & Lipson, H. 'Attribute-Based Architectural Styles," 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture* (WICSA1)). San Antonio, TX: February 1999.