

Identifying Aspects Using Architectural Reasoning

Len Bass
*Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213 USA
+1 412 268-6763
ljb@sei.cmu.edu*

Mark Klein
*Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213 USA
+1 412 268-7615
mk@sei.cmu.edu*

Linda Northrop
*Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213 USA
+1 412 268-7638
lmn@sei.cmu.edu*

1. Introduction

Software architecture, which encompasses the structures of software systems, has emerged as a crucial part of the design process. A software architecture is developed as the first step toward designing a system with certain desired properties. A growing body of experience and evidence suggest that the dominant design drivers for any software architecture are its quality attribute requirements [3]. By quality attributes we mean such properties as reliability, security, usability, modifiability, performance, portability, etc. Quality attribute requirements articulate the important quality attributes for a system in system-specific characterizations. We are developing a method to derive a software architecture from its quality attribute requirements via insights gained from quality attribute models called architectural tactics. We believe that some of the discoveries that occur during the derivation process can be viewed as candidate aspects. In particular, the derivation process illuminates what we call *architectural aspects*. Architectural aspects come with architectural analogous advice, pointcuts, and join points. The architectural aspects are candidate aspects to be carried through detailed design and implemented using AOP.

We describe this connection in a bottom-up fashion because we think it is helpful to see the concepts at work in a simple example before delving into the details of the method. So after we set the stage by introducing some new terminology, we begin with a small set of quality requirements for an example system, present a software architecture that satisfies those requirements, and highlight the architectural tactics at work in that architecture. We then identify architectural aspects and their constituent architectural advice, pointcuts, and join points. Having motivated the need to derive the relevant architectural tactics, we next summarize our method and

how it can derive the tactics and architectural aspects already described. In practice, one would, of course, begin with the quality requirements, apply the method, derive the architecture, and, in the process, identify the candidate aspects. We conclude with a postulation of the benefits associated with aspect identification during architecture design, as well as a discussion of open issues and possible future work.

2. Terminology

Kiczales and colleagues use the following terms in their overview of AspectJ [6]:

- *Join points* are well-defined points in the execution of the program.
- *Pointcuts* are a means of referring to collections of join points and certain values at those join points.
- *Advice* consists of method-like constructs used to define additional behavior at join points.
- *Aspects* are units of modular crosscutting implementation composed of pointcuts, advice, and ordinary Java member declarations.

We use the same terminology extended with a prefix of *architectural* to emphasize that we are applying these aspect concepts to architectural reasoning rather than language reasoning.

- *Architectural join points* are well-defined points in the specification of the software architecture.
- *Architectural pointcuts* are means of referring to collections of architectural join points.
- *Architectural advice* is a specification (possibly informal) of transformations to perform at architectural join points.

- *Architectural aspects* are architectural views consisting of architectural pointcuts and architectural advice.

3. Example

The example we use is a variation of one that we have used before [1, 3], a product line of garage door opener systems. This system is a portion of a home integration system (HIS) that manages the opening and closing of the garage door.

The basic premise of our work in architecture design is that quality attribute requirements are the dominant design drivers for any system. We express quality attribute requirements as scenarios formulated in a structured fashion. The following are the quality attribute requirements for our example. In each case the relevant quality attribute is indicated in square brackets at the end of the requirement.

Reacting to obstacles: The garage door system must be able to detect obstacles and safely halt a closing garage door in a timely manner. Timely is defined to be within 0.1 seconds. Also, when an obstacle is detected, this event must be reported to the user interface (UI) display (if there is one) and to an HIS (if there is one). [performance]

Door commands: The garage door system can receive commands from a remote control in the car, the HIS, or buttons attached to the garage. Several commands must be handled: open, close, halt, and diagnosis. The garage door system must be able to detect a command and initiate its execution within 0.5 seconds. (Note that when “halt” is issued via obstacle detection sensors, it is subject to the .1 second requirement.) [performance]

UI: Various garage door openers have various controls. Some have displays and others do not; some may be integrated with an HIS while other cannot be; various types of remote controls should be supported. [modifiability]

Sensors and actuators: Sensors and actuators can change due to either obsolescence or changes in the marketplace. [modifiability]

4. Architecture of example

Figure 1 depicts the module decomposition view of an architecture for the garage door opener system. It has the following modules with associated responsibilities:

- *UI.* Manage interactions with UI sensors and displays. A change in UI devices will be handled here.
- *Virtual UI.* Translate interactions from the UI into a set of commands common for all UIs. This module hides all changes in UI devices so they are not visible elsewhere in the system. This module also checks security codes by sending a message to the House proxy (through the Pub/Sub module).
- *Pub/Sub.* Handle events received from modules on a subscription basis. Doing so keeps portions of the system that exist in some products and not in others (such as the HIS) from affecting the remainder of the system.
- *House proxy.* Manage interactions with the HIS (if it exists).
- *Door commands.* Manage all the door’s actions.
- *Virtual sensor/actuator.* Hide details of actual sensors or actuators (except for obstacle detection). Doing so allows sensors or actuators to change without impacting the remainder of the system.
- *Device driver.* Handle interactions with particular sensors or actuators except for the obstacle detection sensor.
- *Obstacle detection algorithm.* Handle obstacle detection. It sends a halt instruction directly to the door motor.
- *Obstacle detection driver.* Handle interactions with the obstacle detection sensor.

Notice that a major influence on the module view is the requirement to accommodate change. The modules whose role is to hide the effects of changes include Virtual UI, Virtual sensors/actuators, House proxy, and Pub/Sub.

Figure 2 depicts the concurrency view of the same architecture. It shows some threads and associated communication paths among the components derived from the modules. The numbers in the circles reflect scheduling priority. The Obstacle Detection thread has highest priority, the Door Commands and the House Proxy threads have the next highest priorities, and the Status Reporting thread has the lowest priority. Notice that a major influence on the concurrency view is the need to satisfy stringent timing requirements. In particular, this architecture allows for high-performance obstacle detection.

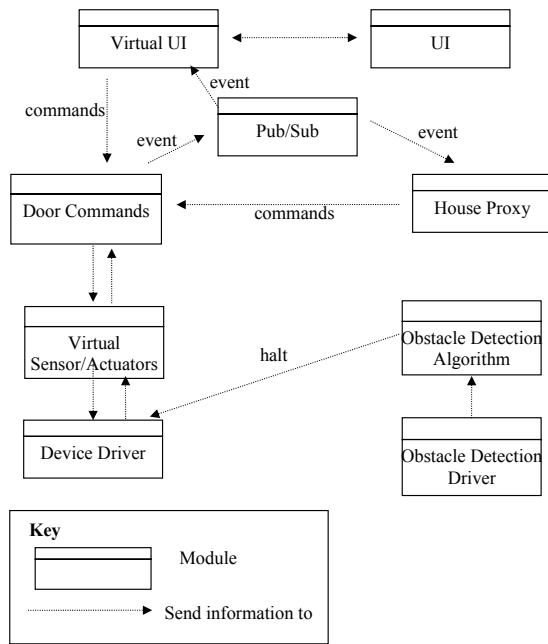


Figure 1. Module view

5. Architectural tactics and aspects

We define an architectural tactic as “a means of satisfying a quality-attribute-response measure by manipulating some aspect of a quality attribute model through architectural design decisions” [1]. We have enumerated architectural tactics for the achievement of modifiability and performance. A subset of the tactics we identified lead directly to architectural aspects.

Two key modifiability tactics are *Abstract common services* and *Break the dependency chain*. The *Abstract common services* tactic prescribes generating a new module comprising common or similar responsibilities that previously resided in multiple modules. This tactic localizes similar responsibilities. The *Break the dependency chain* tactic prescribes inserting an intermediary between the publisher and consumer of data or services. This tactic helps prevent the propagation of a change to the producer from propagating to the consumer (or vice versa). Looking again at Figure 1, we see that the *Abstract common services* tactic was applied to create the Virtual UI and the Virtual sensor/actuator modules, and the *Break the dependency chain* tactic was applied to generate the Pub/Sub module (between the Door commands module and several consumers of its events).

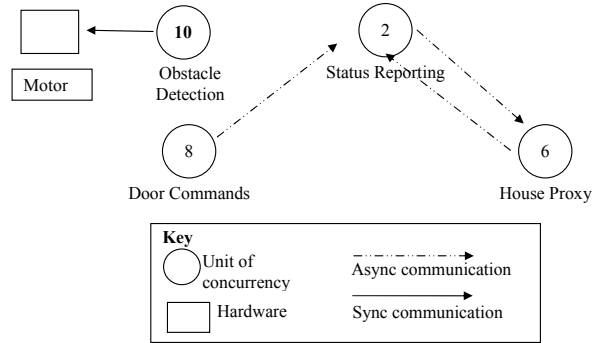


Figure 2. Concurrency view

The *Break the dependency chain* tactic leads directly to an architectural aspect. In this case the aspect implements the crosscutting concerns related to the publisher/subscriber:

- architectural aspect – the collection of pointcuts and advice related to the publisher/subscriber
- architectural pointcuts – specification for every place where an event is published or subscribed to, and every place where an event is defined
- architectural advice – establishes the right connections between producers and consumers of events, and provides the details of the types of generated events

This architectural aspect would allow the designer to defer the choice of the details of the publish/subscribe mechanism for communication, such as the choice of shared memory or message passing, and the subsequent decision would be woven in at either compile time or runtime. If there is no subscriber for certain events (e.g., no House proxy), the aspect can eliminate the generation of messages for those events.

Similarly the Virtual UI and Virtual sensor/actuator modules can be considered architectural aspects. Consider the Virtual sensor/actuator module:

- architectural aspect – the collection of pointcuts and advice related to sensor/actuator services
- architectural pointcuts – specification for every place where a sensor/actuator service is invoked
- architectural advice – specification of the device driver details associated with each generic sensor/actuator service—in effect the entire Device driver module.

One performance tactic evident in the concurrency view is *Use fixed-priority scheduling*. This tactic, which leads directly to an architectural aspect, assumes that each unit of concurrency (see Figure 2) is assigned a fixed priority.

- architectural aspect – the collection of pointcuts and advice related to the prioritization of units of concurrency
- architectural pointcuts – specification for every place where the priority of a unit of concurrency is specified
- architectural advice – specification of the rules inspecting the relative deadlines of units of concurrency and using those rules as the basis for assigning priorities

Making the priority assignment an architectural advice allows the priorities to be modified in a uniform fashion and modularizes this crosscutting concern.

Another performance tactic is *Increase logical concurrency*, which is likewise an architectural aspect.

- architectural aspect – the collection of pointcuts and advice related to communication between units of concurrency
- architectural pointcuts – specification for every place where communication services are used
- architectural advice – specification of the communication protocol—asynchronous or synchronous—based on the communication path

This architectural aspect localizes the communication strategy so that it can be changed uniformly.

The key point to observe from this discussion is that each use of an architectural tactic is manifested by particular points in the architecture, and each point is a candidate for an architectural join point. These join points would be controlled by the architectural advice. Whether an architectural aspect should be used to depict any given tactic is something for which we have, as yet, no general rules. At this point, what we can say is that each application of a tactic is a candidate for an architectural aspect. Moreover, treating the tactic as an architectural aspect seems to bring to the architecture the same advantages that aspects bring to programming.

The use of an architectural tactic is not a random event but rather the result of a deliberate design decision. In the next section, we describe how tactics are chosen.

6. Method for identifying tactics

The example in the previous section illustrates that the notion of aspects is applicable at the architecture level and that architectural aspects are discovered through the identification of the architectural tactics being applied. The obvious question that arises is: how does one go about identifying the appropriate architectural tactics to apply? We developed a method that shows how to derive architectural decisions from a set of quality attribute requirements through the application of architectural

tactics. We can extend this method easily to identify candidate architectural aspects.

Figure 3 shows the key elements of our method. Quality requirements are expressed as quality attribute scenarios. In the figure we show only modifiability and performance requirements. For the purposes of our method, functional requirements are expressed in a variety of forms such as text, state charts, use cases, and then are translated into a responsibility graph.

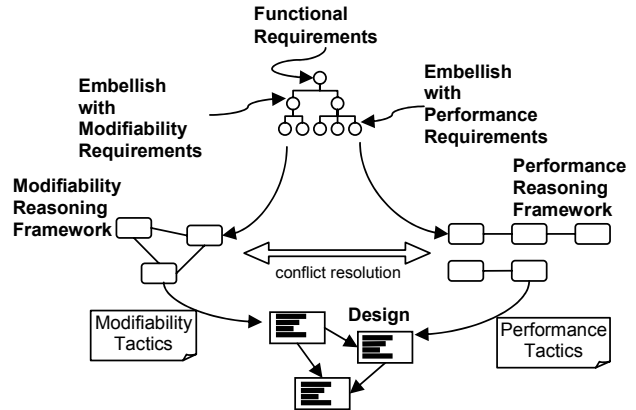


Figure 3. Elements used in our method

6.1 Formulate quality attribute requirements

We formulate quality attribute requirements according to a structured scenario format made up of these six parts: 1. stimulus to which the scenario reacts. This can be at runtime or prior to runtime, depending on the quality attribute; 2. the source of the stimulus; 3. the artifact affected by the stimulus; 4. the environment of the system when the stimulus arrives; 5. the desired response to the stimulus; 6. the measure for the response and the bound (constraint) on that measure that must be satisfied

Table 1 gives the parts and the values for the three scenarios we are considering.

Table 1. Scenarios being considered

| | Obstacle detection | Modify sensor/actuator except motor | Modify motor |
|--------------------|---------------------------|--|--|
| Source of stimulus | Obstacle detection sensor | New product request | New product request |
| Stimulus | Obstacle detected | Develop new product with different sensor/actuator | Develop new product with different motor |
| Environment | Garage door | Development | Development |

| | | | |
|--|----------------|------------------------|------------------------|
| | descending | time | time |
| Artifact | Motor | Source code for system | Source code for system |
| Response | Halt door | Produce new product | Produce new product |
| Response measure and constraint | Within .1 sec. | Within 1 staff-days | Within 1.5 staff-days |

For each scenario the relevant reasoning framework is deducible from its response measure. The first scenario has a timing deadline, indicating that a reasoning framework that deduces latency would be appropriate; the Rate Monotonic Analysis [5] is therefore suitable. The other two scenarios have staff-days as the response measure, indicating that the Impact Analysis reasoning framework should be used to reason about the achievement of these scenarios.

6.2 Construct quality attribute model

We begin with the Rate Monotonic Analysis reasoning framework.

6.2.1 Rate Monotonic Analysis

The Rate Monotonic Analysis reasoning framework suggests an assignment of responsibilities to units of concurrencies and uses estimates of the execution time and the priorities of the units of concurrency to determine the latency in response to a particular event.

The architect must estimate the execution times for the responsibilities. During the action of the reasoning framework, the architect may be asked whether it is possible to reduce those estimates. After one successful satisfaction of the scenarios, the execution time estimates become budgets that must be adhered to by subsequent implementation.

Some of the architectural tactics comprising the Rate Monotonic Analysis reasoning framework for this example include

- *Reduce execution time.* This tactic causes the software architect to consider what the execution times are for each responsibility and whether they can be reduced.
- *Use fixed-priority scheduling.* This tactic determines the basis for assigning priorities and calculating the latency in response to the single event of obstacle detection.

- *Increase logical concurrency.* In our example, we used this tactic as a method for reducing the latency associated with responsibilities that have stringent timing requirements.

For example, the *Use fixed priority scheduling* tactic suggests assigning the responsibilities “detect obstacle” and “halt door” to the highest priority unit of concurrency, since those responsibilities have the most stringent timing requirement associated with them. Rate Monotonic Analysis gives us rules of thumb for assigning priorities (shortest deadline first) that will satisfy the constraints (if all the performance constraints can be satisfied). The Rate Monotonic Analysis reasoning framework, through the use of tactics, focuses the architect’s attention on the design decisions that are the most influential in achieving our performance goals. As we saw in the previous section, architectural aspects provide us with a way to modularize these high-impact architectural decisions.

6.2.2 Impact analysis

The Impact Analysis [4] reasoning framework assigns responsibilities to modules in such a way as to satisfy the constraints from the scenarios. The reasoning framework may also refine the responsibilities according to various tactics.

The Impact Analysis reasoning framework constructs a dependency graph among modules. The nodes of the graph are modules, and the arc between nodes A and B represents the probability that a modification to node A will require a derivative modification to node B. For a given dependency graph, a cost function determines the cost of a modification to a particular responsibility within a given node. See [2] for details of the cost model; we will not go into its workings here. The cost model depends on the cost of modifying a responsibility, the packaging of the responsibilities into modules, and the probability of a modification of one responsibility propagating to another.

In the Rate Monotonic Analysis reasoning framework, the architect needed to specify the execution time of responsibilities, the assignment of responsibilities to units of concurrency, and the priority of those units of concurrency.

In the Impact Analysis reasoning framework, the architect must specify the cost of modifying each responsibility. Doing so is equivalent to specifying the execution time for the Rate Monotonic Analysis reasoning framework. The architect specifies the cost using experience and intuition—just as he or she specifies

the execution time. One tactic encourages the architect to think about the specified cost of modifying a responsibility and whether it can be reduced, again in parallel with Rate Monotonic Analysis.

There are two types of tactics within the Impact Analysis reasoning framework. One type is intended to reduce the cost of a modification to a responsibility by modifying the set of responsibilities, and one type is intended to reduce the likelihood that a modification is propagated to other responsibilities.

In our example, two scenarios are relevant to the Impact Analysis reasoning framework—replace sensors or actuators, and replace the motor.

The reasoning framework begins by considering the cost of modifying the software to manage a new sensor. If this cost is less than the constraint value, this scenario is satisfied. For our example, we assume that the constraint is not met and therefore that tactics need to be applied.

The first set of tactics to consider are those that attempt to reduce the cost of making a modification. The *Abstract common services* tactic suggests to the architect that the management of sensors can be considered a common service. This approach causes a potential rearrangement of responsibilities into those that manage the sensors (the sensor device drivers) and those that use the sensor management responsibilities. Now the cost of changing the sensor is recalculated using the reasoning framework. Once again, we assume the cost is too high. The architect examines the cost model and sees that changes to the sensor cause changes to the sensor device driver that, in turn, are propagated through the remainder of the responsibilities. This observation causes the architect to apply the *Break the ripple effect* tactic. Again the responsibilities are rearranged so that a virtual sensor/actuator is interposed between the device drivers and the other responsibilities. This time, the cost of a change is examined and the constraint is now met.

Notice that the application of the tactics is a joint undertaking of the method and the architect. The method will highlight where problems occur and which tactics might be applicable. The architect then chooses appropriate tactics and how they are applied. Since nothing, thus far, has affected the scheduling of the obstacle detection, the performance constraint is still satisfied.

Now, the other modifiability scenario, one that calls for the modification of the motor, is examined. The same initial reasoning that led to sensor/actual device drivers leads to a motor device driver. This time, however, the constraint is 1.5 staff-days rather than 1 staff-day. This larger constraint is satisfied just by the device driver without interposing a virtual motor.

The Impact Analysis reasoning framework, through the use of tactics, focuses the architect's attention on the design decisions that are the most influential in achieving modifiability goals. As we saw for performance, aspects also provide us with a way of modularizing the high-impact architectural decisions that affect modifiability.

6.3 Convey information among reasoning frameworks

In addition to those responsibilities extracted from the scenarios, the other responsibilities for the garage door system must be enumerated. For our purposes, they are lumped into a single responsibility called "other."

The only vocabulary that the Rate Monotonic Analysis reasoning framework and the Impact Analysis reasoning framework have in common is the concept of responsibilities. A responsibility graph is used to capture the relationships among the responsibilities. Possible relationships are either is-a-part-of or precedes. Responsibilities can be decomposed to yield is-a-part-of relationships. This is what happens when the common services are embodied in the Virtual sensor/actual module. Responsibilities can also precede other responsibilities. For example, an obstacle must be detected prior to the door being halted in one of our scenarios.

Both reasoning frameworks, during their activities, drew on the responsibility graph to determine responsibilities for allocation to either units of concurrency or modules, and modified the responsibility graph through the addition of new responsibilities or the decomposition of responsibilities.

6.4 Satisfy conflicting constraints

In general, a reasoning framework considers a set of possible tactics (derived from looking at the analysis model that the reasoning framework uses) and attempts to find a combination of tactics (in conjunction with the architect) that will satisfy all the relevant constraints. Each application of a tactic may affect the responsibility graph that may, in turn, affect reasoning frameworks other than the one that is applying the tactic. Recalling our prior discussion, application of some tactics also identifies a candidate architectural aspect.

When the various data structures are stable, the implied design should be generated. If the data structures do not stabilize or if some constraints are unable to be satisfied, the system is overconstrained and cannot be constructed. In this case, either a constraint must be

relaxed or parameters (such as execution time) are adjusted.

6.5 Interaction between the design method and the architect

We did not show the derivation that led to two scenarios to reflect the cost of modification. Originally, there was only one scenario that dealt with the modification of sensors/actuators. That scenario led, using a chain of reasoning as above, to a Virtual sensor/actuator module. The performance constraint involving the motor, however, could not be met using a virtual motor. Thus, the architect was informed, through the method, that the quality requirements could not be met. This caused the architect to weaken the modifiability requirement for the motor.

Although the method provides guidance and information to the architect, he or she still makes the key decisions.

7. Conclusions and open issues

We have presented a method that begins with quality requirements cast as quality scenarios and moves from there (in a semi-automatic fashion) to a design. The method is based on using quality attribute reasoning frameworks to determine whether the requirements can be met, to identify places in an emerging design that cause them not to be met, and to provide the architect with guidance as to which architectural tactics to use to correct the problem.

We initially used an example to present our method from the bottom up for pedagogical purposes. From the top down, the method proceeds as follows: Represent quality attribute requirements as structured scenarios; Generate a design using reasoning frameworks and architectural tactics that depend on those reasoning frameworks to satisfy quality attribute requirements; Consider each use of an architectural tactic in generating the design as a candidate architectural aspect, where the join point is the place in the architecture where the tactic was applied and the advice is dependent on the quality attribute being achieved by the tactic.

This leaves two questions for future consideration:

- Which subset of architectural tactics leads to architectural aspects?

- Which subset of architectural aspects is realizable as language-level aspects?

We believe there is a systematic method for addressing both questions, but they remain open issues.

Some other ideas that we believe have promise at the architectural level and merit investigation include

- identifying an “aspect” view of the architecture that would display those places in the architecture where decisions have been deferred and where an “architectural aspect” weaver might be useful
- Many of the aspects we have identified require a weaver to operate at more than just a syntactic level. For example, a weaver may decide based on performance considerations identified by a method such as ours that for one join point, information can be passed as messages but for another, it should be passed through shared memory.
- some method for specifying functionality and architectural style independently and weaving them together

8. Acknowledgments

Our thanks to Gregor Kiczales, Gail Murphy, and Harold Osher for participating in a workshop at the Software Engineering Institute that led to this paper.

9. References

- [1] Felix Bachmann, Len Bass, and Mark Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design (CMU/SEI-2003-TR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003.
- [2] Len Bass, Felix Bachmann, and Mark Klein. Making Variability Decisions During Architecture Design, Proceedings Product Family Engineering – 5, Nov, 2003, Sienna, Italy, Springer-Verlag.
- [3] Len Bass, Rick Kazman, and Paul Clements. Software Architecture in Practice, 2nd edition. Reading, MA: Addison-Wesley, 2003.
- [4] Shawn A. Bohner and Robert S. Arnold. Software Change Impact Analysis. IEEE Computer Society Press, 1996.
- [5] J.P. Lehoczky, “Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines,” Proc. of the 11th IEEE Real-Time Systems Symposium, pp. 201-209, December 1990.
- [6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ In Proceedings of ECOOP, Springer-Verlag (2001).