# The Architecture Tradeoff Analysis Method.

**6 authors**, including:

Rick Kazman
Carnegie Mellon University
**370** PUBLICATIONS   **19,485** CITATIONS

Thomas Longstaff
Johns Hopkins University
**44** PUBLICATIONS   **6,190** CITATIONS

Howard F. Lipson
Carnegie Mellon University
**25** PUBLICATIONS   **1,928** CITATIONS

Jeromy Carriere
**26** PUBLICATIONS   **2,479** CITATIONS

# The Architecture Tradeoff Analysis Method

Rick Kazman
Mark Klein
Mario Barbacci
Tom Longstaff
Howard Lipson
Jeromy Carriere

*July 1998*

# The Architecture Tradeoff Analysis Method

Rick Kazman

Mark Klein

Mario Barbacci

Tom Longstaff

Howard Lipson

Jeromy Carriere

*July 1998*

**Product Line Systems**

# Table of Contents

# List of Figures

# List of Tables

# Abstract

This paper presents the Architecture Tradeoff Analysis Method (ATAM), a structured technique for understanding the tradeoffs inherent in the architectures of software-intensive systems. This method was developed to provide a principled way to evaluate a software architecture's fitness with respect to multiple competing quality attributes: modifiability, security, performance, availability, and so forth. These attributes interact, and improving one often comes at the price of worsening one or more of the others. The method helps us reason about architectural decisions that affect quality attribute interactions. The ATAM is a spiral model of design, one of postulating candidate architectures followed by analysis and risk mitigation that lead to refined architectures.

# 1    Architecture Tradeoff Analysis

Quality attributes of large software systems are principally determined by the system's software architecture. That is, in large systems, the achievement of qualities such as performance, availability, and modifiability depends more on the overall software architecture than on code-level practices such as language choice, detailed design, algorithms, data structures, testing, and so forth. This is not to say that the choice of algorithms or data structures is unimportant, but rather that such choices are less crucial to a system's success than its overall software structure, its architecture. Thus, it is in our interest to try and determine, before it is built, whether a system is destined to satisfy its desired qualities.

Although methods for analyzing specific quality attributes exist (e.g., [Bass 98], [Kazman 96], [Klein 93], [Smith 93]), these analyses have typically been performed in isolation. In reality, however, the attributes of a system *interact*. Performance impacts modifiability. Availability impacts safety. Security affects performance. Everything affects cost and so forth. While experienced designers know that these tradeoffs exist, there is no principled method for characterizing them and, in particular, for characterizing the interactions among attributes.

For this reason, software architectures are often designed "in the dark." Tradeoffs are made—they must be made if the system is to be built—but they are made in an *ad hoc* fashion. Imagine a sound engineer being given a 28-band graphic equalizer, where each of the equalizer's controls has effects that interact with some subset of the other controls. But the engineer is not given a spectrum analyzer, and is asked to set up a sound stage for optimal fidelity. Clearly such a task is untenable. The only difference between this analogy and software architecture is that software systems have far more than 28 independent but interacting variables to be "tuned."

There are techniques that designers have used to try to mitigate the risks in choosing an architecture to meet a broad palette of quality attributes. The recent activity in cataloguing design patterns and architectural styles is an example of this. A designer will choose one pattern because it is "good for portability" and another because it is "easily modifiable." But the analysis of patterns doesn't go any deeper than that. A user of these patterns does not know how portable, modifiable, or robust an architecture is until it has been built.

To address these problems, this paper introduces the Architecture Tradeoff Analysis Method (ATAM). ATAM is a method for evaluating architecture-level designs that considers multiple quality attributes such as modifiability, performance, reliability, and security in gaining insight

as to whether the fully fleshed out incarnation of the architecture will meet its requirements. The method identifies tradeoff points between these attributes, facilitates communication between stakeholders (such as user, developer, customer, maintainer) from the perspective of each attribute, clarifies and refines requirements, and provides a framework for an ongoing, concurrent process of system design and analysis.

The ATAM has grown out of work done at the Software Engineering Institute (SEI) on the architectural analysis of individual quality attributes. It began with the Software Architecture Analysis Method (SAAM) [Kazman 96] for modifiability, performance analysis [Klein 93], availability analysis, and security analysis. SAAM has already been used successfully to analyze architectures from a wide variety of domains: software tools, air traffic control, financial management, telephony, multimedia, embedded vehicle control, and so on.

The ATAM, as with the SAAM, has both social and technical aspects. The technical aspects deal with the kinds of data to be collected and how it is analyzed. The social aspects deal with the interactions among the system's stakeholders and area-specific experts, allowing them to communicate using a common framework, to make the implicit assumptions in their analyses explicit, and to provide an objective basis for negotiating the inevitable architecture tradeoffs. This paper will demonstrate the use of the method, and its benefits in clarifying design issues along multiple attribute dimensions, particularly the tradeoffs in design.

# 2 Why Use Architecture Tradeoff Analysis?

All design, in any discipline, involves tradeoffs; this is well accepted. What is less well understood is the means for making informed and possibly even optimal tradeoffs. Design decisions are often made for non-technical reasons: strategic business concerns, meeting the constraints of cost and schedule, using available personnel, and so forth.

Having a structured method helps ensure that the right questions will be asked *early*, during the requirements and design stages when discovered problems can be solved cheaply. It guides users of the method—the stakeholders—to look for conflicts in the requirements and for resolutions to these conflicts in the software architecture.

In realizing the method, we assume that attribute-specific analyses are *interdependent*, and that each quality attribute has connections with other attributes, through specific architectural *elements*. An architectural element is a component, a property of the component, or a property of the relationship between components that affects some quality attribute. For example, the priority of a process is an architectural element that could affect performance. The ATAM helps to identify these dependencies among attributes: what we call *tradeoff points*. This is the principal difference between the ATAM and other software analysis techniques—that it explicitly considers the *connections* between multiple attributes, and permits principled reasoning about the tradeoffs that inevitably result from such connections. Other analysis frameworks, if they consider connections at all, do so only in an informal fashion, or at a high level of abstraction (e.g., [McCall 94], [Smith 93]). As we will show, tradeoff points arise from architectural elements that are affected by multiple attributes.

# 3    The ATAM

The ATAM is a spiral model of *design* [Boehm 86], depicted in Figure 3-1. The ATAM is like the standard spiral model in that each iteration takes one to a more complete understanding of the system, reduces risk, and perturbs the design. It is unlike the standard spiral in that no implementation need be involved; each iteration is motivated by the results of the analysis and results in new, more elaborated, more informed designs.

Analyzing an architecture involves manipulating, controlling, and measuring several sets of architectural elements, environment factors, and architectural constraints. The primary task of an architect is to lay out an architecture that will lead to system behavior which is as close as possible to the requirements within the cost constraints. For example, performance requirements are stated in terms of latency and/or throughput. However, these attributes depend on the architectural elements pertaining to resource allocation: the policy for allocating processes to processors, scheduling concurrent processes on a single processor, or managing access to shared data stores. The architect must understand the impact of such architectural elements on the ability of the system to meet its requirements and manipulate those elements appropriately.
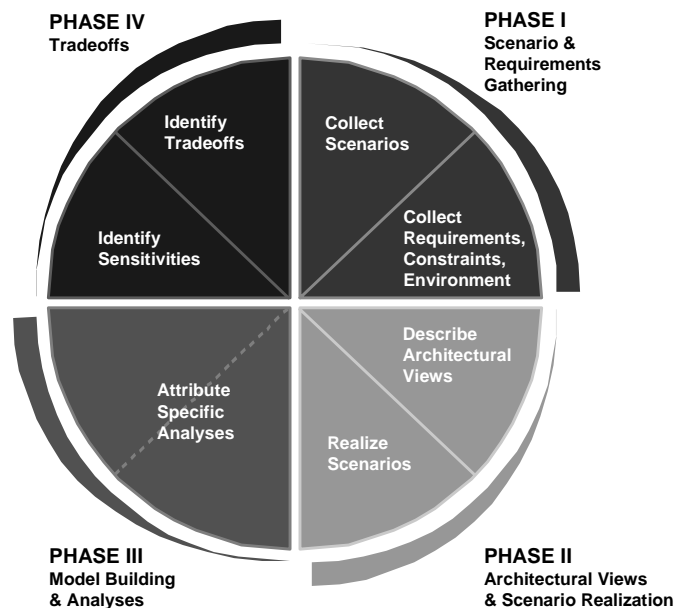


*Figure 3-1:    Steps of the Architecture Tradeoff Analysis Method*

However, this task is typically approached with a dearth of tools. The best architects use their hunches, their experience with other systems, and prototyping to guide them. Occasionally, an explicit modeling step is also included as a design activity, or an explicit formal analysis of a single quality attribute is performed.

## 3.1   The Steps of the Method

The method is divided into four main areas: scenario and requirements gathering, architectural views and scenario realization, model building and analysis, and tradeoffs. Generally, the method works as follows: Once a system's initial set of scenarios and requirements has been elicited and an initial architecture or small set of architectures is proposed (subject to environment and other considerations), each quality attribute is evaluated in turn, and *in isolation*, with respect to any proposed design. After these evaluations comes a critique step. During this step, tradeoff points—elements that affect multiple attributes—are found. After the critique, we can either: refine the models and reevaluate; refine the architectures, change the models to reflect these refinements and reevaluate; or change some requirements. We now look at each of these steps in more detail.

Step 1 — Collect Scenarios
The first and second steps in the method—eliciting system usage scenarios and collecting requirements, constraints, and environmental details from a representative group of stakeholders—are interchangeable. At times requirements exist before any architectural analysis has commenced. At other times scenarios drive the requirements. The analysis process can profitably begin with either. Eliciting scenarios serves the same purposes as it does in SAAM: to operationalize both functional and quality requirements, to facilitate communication between stakeholders, and to develop a common vision of the important activities the system should support.

Step 2 — Collect Requirements/Constraints/Environment
In addition to the scenarios, the *attribute-based* requirements, constraints, and environment of the system must be identified. A requirement can have a specific value or can be more generic, as derived from scenarios of hypothetical situations. The environment must be characterized because subsequent analyses (e.g., performance or security) and constraints on the design space, as they evolve, are recorded as these too affect attribute analyses. This step places a strong emphasis on revisiting the scenarios from the previous step to ensure that they account for important quality attributes.

Step 3 — Describe Architectural Views
The requirements, scenarios, and engineering design principles together generate candidate architectures and constrain the space of design possibilities. In addition, design almost never starts from a clean slate: legacy systems, interoperability, and the successes/failures of previous projects all constrain the space of architectures.

The candidate architectures must be described in terms of the architectural elements and properties that are relevant to each of the important quality attributes. For example, replication and voting schemes are an important element for reliability; concurrency decomposition, process priorities, throughput estimates, and queuing schemes are important for performance; firewalls and intruder models are important for security; abstraction and encapsulation is important for modifiability. In addition, the details required for an analysis of each of these qualities is typically captured in distinct *architectural views*. Bass et al [Bass 98] define several such views for use in architectural analysis: a module view (for reasoning about work assignments and information hiding), a process view (for reasoning about performance), a dataflow view (for reasoning about functional requirements), a class view (for reasoning about sharing of object definitions), and so forth.

There is another important point to make about architectural representation: in our presentation of the ATAM, we assume that *multiple, competing* architectures are being compared. However, designers typically consider themselves to be working on only a *single* architecture at a time. Why are these notions not aligned? From our perspective, an architecture is a collection of functionality assigned to a set of structural elements, described by a set of views. Almost any change will mutate one of these views, thus resulting in a new architecture. While this point might seem like a splitting of hairs, it is important hairs in this context for the following reason: the ATAM requires building and maintaining attribute models (both quantitative and qualitative models) that reflect and help to reason about the architecture. To change any aspect of an architecture—functionality, structural elements, coordination model—will affect one or more of the models. Once a change has been proposed, the new and old architectures are "competing," and must be compared: hence the need for new models that mirror those changes. Using the ATAM, then, is a continual process of choosing among competing architectures, even when these look "pretty much the same" to a casual observer.

Step 4 — Attribute-Specific Analyses

Once a system's initial set of requirements and scenarios has been elicited and an initial architecture (or small set of architectures) is proposed, each quality attribute must be analyzed *in isolation*, with respect to each architecture. These analyses can be conducted in any order; no individual critique of attributes against requirements or interaction between attributes is done at this point. Allowing separate (concurrent) analysis is an important separation of concerns that allows individual attribute experts to bring their expertise to bare on the system.

The result of the analyses leads to statements about system behavior with respect to values of particular attributes: "requests are responded to in 60 ms. average"; "the mean time to failure is 2.3 days"; "the system is resistant to known attack scripts"; "the hardware will cost $80,000 per platform"; "the software will require 4 people per year to maintain"; and so forth.

Step 5 — Identify Sensitivities
This step determines the sensitivity of individual attribute analyses to particular architectural elements; that is, one or more attributes of the architecture are varied. The models are then varied to capture these design changes, and the results are evaluated. Any modeled values that are significantly affected by a change to the architecture are considered to be *sensitivity points*.

Step 6 — Identify Tradeoffs
The next step of the method is to critique the models built in Step 4 and to find the architectural tradeoff points. Although it is standard practice to critique designs, significant additional leverage can be gained by focusing this critique on the *interaction* of attribute-specific analyses, particularly the location of tradeoff points. Here is how this is done.

Once the architectural sensitivity points have been determined, finding *tradeoff* points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The availability of that architecture might also vary directly with the number of servers. However, the security of the system might vary inversely with the number of servers (because the system contains more potential points of attack). The number of servers, then, is a tradeoff point with respect to this architecture. It is an element, potentially one of many, where architectural tradeoffs will be made, consciously or unconsciously.

## 3.2   Iterations of the ATAM

After we have completed the above steps, we can compare the results of the analyses to the requirements. When the analyses show that the system's predicted behavior comes adequately close to its requirements, the designers can proceed to a more detailed level of design or to implementation. In practice, however, it is useful to continue to track the architecture with analytic models to support development, deployment, and maintenance. Design never ceases in a system's life cycle, and neither should analysis.

In the event that the analysis reveals a problem, we now develop an action plan for changing the architecture, the models, or the requirements. The action plan will draw on the attribute-specific analyses and identification of tradeoff points. This then leads to another iteration of the method.

It should be made clear that we do not expect these steps to be followed linearly. They can and do interact with each other in complex ways: an analysis can lead to the reconsideration of a requirement; the building of a model can point out places where the architecture has not been adequately thought out or documented. This is why we depict the steps as wedges in a circle: at the center of the circle every step touches (and exchanges information with) every other step.

# 4 An Example Analysis

To exemplify the ATAM, we have chosen an example that has already been analyzed extensively in the research literature, that of a remote temperature sensor (discussed in [Smith 93] and elsewhere). We have chosen this example precisely because it has been heavily scrutinized already. The existence of other analyses focuses attention on the differences of the ATAM. We will analyze this system with respect to its availability, security, and performance attributes.

## 4.1 System Description

The RTS (remote temperature sensor) system exists to measure the temperatures of a set of furnaces, and to report those temperatures to an operator at some other location. In the original example, the operator was located at a "host" computer. The RTS sends *periodic temperature updates* to the host computer, and the host computer sends *control requests* to the RTS, to change the frequency at which periodic updates are sent. These requests and updates are done on a furnace by furnace basis. That is, each furnace can be reporting its temperature at a different frequency. The RTS is presumably part of a larger process control system. The control part of the system is not discussed in this example, however.

We are interested in analyzing the RTS for the qualities of performance, security, and availability. To illustrate these analyses, we have made the model problem richer and more complex than its original manifestation. In addition to the original set of functional requirements, we have embedded the RTS into a system architecture based on the client-server idiom. The remote temperature sensor functionality is encapsulated in a server that serves some number of clients. To remain consistent with the original problem, our analysis will assume that there are 16 clients: one per furnace.

The RTS server hardware includes an analog-to-digital converter (ADC), that can read and convert a temperature for one furnace at a time. Requests for temperature readings are queued and fed, one at a time, to the ADC. The ADC measures the temperature of each furnace at the frequency specified by its most recently received control request.

# 5    Collect Scenarios

The performance analysis is guided by two scenarios, representing the two most common uses of the system:

- the client sends a control request and receives the first periodic update
- the client receives periodic updates at the specified rate

The availability analysis is also guided by two scenarios, representing anticipated ways in which the system could fail:

- a server suffers a software failure and is rebooted
- a server suffers a power supply failure and the power supply is replaced

# 6 Collect Requirements/Constraints/ Environment

In a performance analysis, we consider requirements that typically are derived from scenarios generated through interviews with the stakeholders. In this case, the performance requirements are:

> PR1: Client must receive a temperature reading *within F seconds* of sending a control request.

> PR2: Given that Client X has requested a periodic update every T(i) seconds, it must receive a temperature *on the average every T(i) seconds*.

> PR3: The interval between consecutive periodic updates must be *not more than 2T(i) seconds*.

In addition to these requirements, we will assume that the behavior patterns and execution environment are as follows:

- relatively infrequent control requests
- requests are not dropped
- no message priorities
- server latency = de-queuing time ($Cdq$ = 10 ms) + furnace task computation ($Cfnc$ = 160 ms)
- network latency between client and server ($Cnet$ = 1200 ms)

For the availability analysis, we will initially consider only a single requirement for the RTS system:

> AR1: System must not be unavailable for more than 60 minutes per year.

Our sole environmental assumption is that the availability analysis should also consider a range of component failure rates, from 0 to 24 per year.

Because attributes "trade off" against each other, each of these assumptions is subject to inspection, validation, and questioning as a result of the ATAM.

# 7    Describe Architectural Views

Since the ATAM was created to illustrate architectural tradeoffs, we need some architectures to analyze. We will consider three architectural options and will show a run-time *process* view of each: a simple *Client-Server* architecture, a more complex version of this architecture (called *Client-Server-Server*) where the server has been replicated, and finally an option called *Client-Intelligent Cache-Server*. Each of these architectures will use the identical *server* architecture—all that changes are the ways in which the rest of the system interacts with the server (or servers).
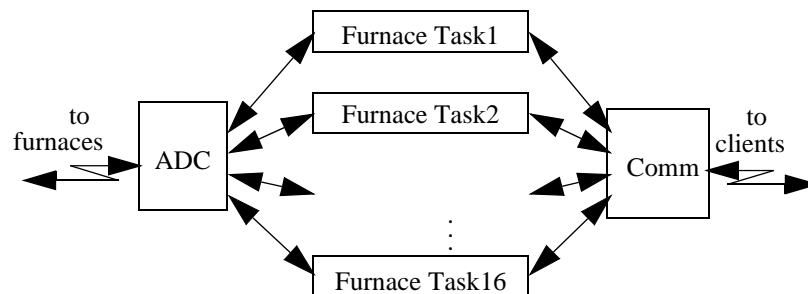


*Figure 7-1:    The Architecture of a Furnace Server*

The server's architecture (shown in Figure 7-1) contains three kinds of components: furnace tasks (independently scheduled units of execution), that schedule themselves to run with some period; a shared communication facility task, that accepts messages from the furnace tasks and sends them to a specified client; and the ADC task, which accepts requests from the furnace tasks, interfaces with the physical furnaces to determine their temperatures, and passes the result back to the requesting furnace task.

Now that the server architecture has been described, we will present the overall system architectures. In each system, a set of 16 clients interacts with one or more servers, communicating via a network.

## 7.1 Architectural Option 1 (Client-Server)

Option 1 (shown in Figure 7-2) is the baseline: a simple and inexpensive client-server architecture, with a single server serving all 16 clients.
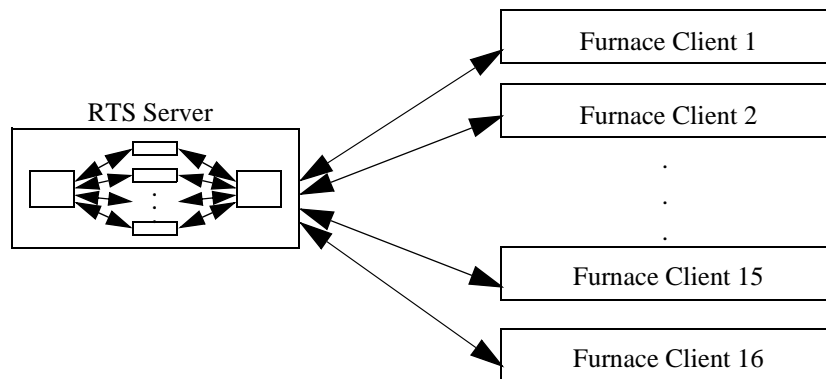


*Figure 7-2:     Option 1's Architecture*

## 7.2 Architectural Option 2 (Client-Server-Server)

Option 2 differs from option 1 in that it adds a second server to the system architecture. These servers interact with clients as a "primary" server (indicated by the solid lines between servers and clients) or as a "backup" server (indicated by the dashed lines). As shown in Figure 7-3, each server has its own set of independent furnace tasks, ADC and Comm, but communicates with the same furnaces and with the same set of clients, although under normal operation each server only serves 8 of the 16 clients.

Every client knows the location of both servers, and if they detect that the server is down (because it has failed to respond for a prescribed period of time), they will automatically switch to their specified backup.
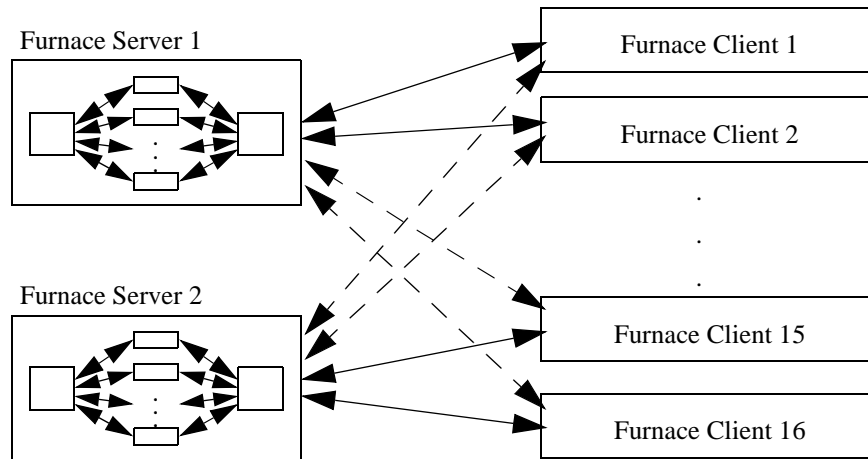


*Figure 7-3:    Option 2's Architecture*

## 7.3    Architectural Option 3 (Client-Intelligent Cache-Server)

Option 3 differs from option 1 in only one way: each client has a "wrapper" that intercedes between it and the server. This wrapper is an "intelligent cache," shown as IC in Figure 7-4. The cache works as follows: it intercepts periodic temperature updates from the server to the client, builds a history of these updates, and then passes each update to the client. In the event of a service interruption, the cache synthesizes updates for the client. It is an *intelligent* cache because the updates it provides take advantage of historical temperature trends to extrapolate plausible values into the future. This intelligence may be nothing more than linear extrapolation, or it might be a sophisticated model that analyzes changes in temperature trends or takes advantage of domain-specific knowledge on how furnaces heat up and cool down.

As long as the furnaces exhibit regular behavior in terms of temperature trends, the cache's extrapolated updates will be accurate. Obviously, the cache's synthesized updates will become less meaningful over time.
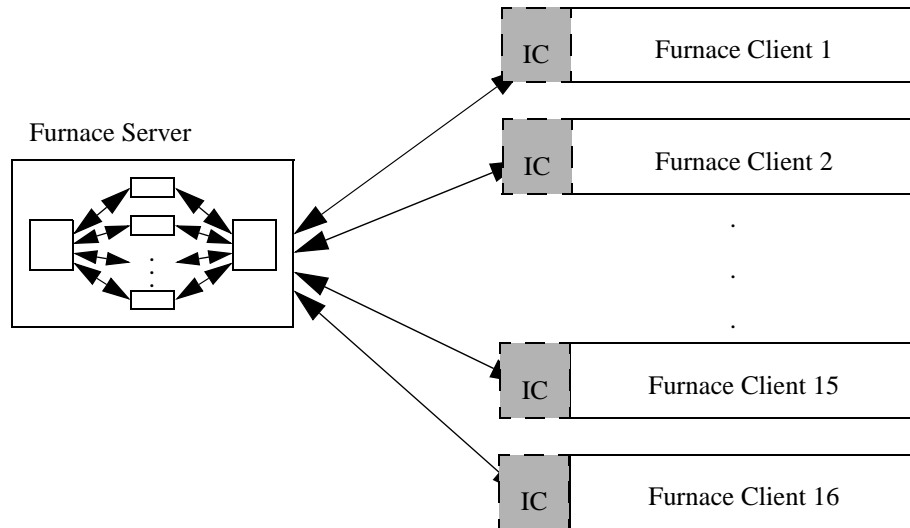


*Figure 7-4:*     *Option 3's Architecture (with Cache)*

These then are our three initial architectural alternatives. To understand and compare them, we will analyze them using the ATAM. This method will aid us in understanding not only the relative strengths and weaknesses of each architecture, but will also provide a structured framework for eliciting and clarifying requirements. This is because each analysis technique incorporates (often implicit) assumptions. The use of several analysis techniques together helps to uncover these assumptions and make them explicit.

# 8  Realize Scenarios/Performance Analyses

To do the performance analysis, we first *realize* the performance scenarios by mapping them onto the software architecture and then calculate whether this realization of the scenario meets the requirements set forth in Section 6.

In the analyses that follow, we will not show the details of doing performance, availability, or security analyses in detail. We do this for two reasons. First, these details can be found in [Barbacci 97]. Second, this paper is not intended to propose or exemplify any particular attribute-specific analysis technique. Indeed, any technique that meets the information requirements of the ATAM would do just as well. Our interest is in how the techniques interact, and how this interaction minimizes risk in a rational, documented design process.

## 8.1  Performance Analysis of Option 1

The performance characteristics of architectural option 1 are summarized in Table 8-1.[1]

| WCCL | ACPL | Jitter |
|------|------|--------|
| 41,120 ms | 5,100 ms | 20,400 ms |

*Table 8-1:  Performance Analysis for Option 1*

A worst-case control latency of 41.12 seconds sounds like a bad thing. However, is it? To answer this question, we must understand the requirement better. How often will the worst case occur? Is it *ever* tolerable to have the worst case occur? For a safety-critical application, the answer might be "no." For an interactive World Wide Web-based application, the answer might be "yes," because the price of ensuring a smaller worst case is prohibitive. Doing an analysis of a single quality attribute forces one to consider such requirements issues.

The worst-case periodic latency is 37.12 seconds. However, the worst-case scenario is unlikely; it assumes that all furnaces are queried at the maximum rate ($T(1) = 10$), that all periodic updates are issued simultaneously, and that the update being measured (the worst-case update) is the last one in the queue. More importantly, in this application the cost of a missed update is not great—another one will arrive in the next $T(i)$ seconds. Given these facts, we cal-

---

1.   WCCL = worst-case control latency; ACPL = average-case periodic latency

culate the *average-case* latency, to see if the system can meet its deadlines under more normal conditions, and accept the fact that an occasional periodic update might be missed.

Finally, we turn to PR3, the "Jitter" requirement. Jitter is the amount of time variation that can occur between consecutive periodic temperature updates. The requirement is that the jitter be not more than 2T(i), which translates to 20 seconds for the case where T(i) = 10. The interval between consecutive readings will be not more than 2T(i) if the difference between best case and worst case latency is not more than 2T(i), for this is an expression of jitter. So, the worst case jitter = $WCPL$[1] - $BCPL$ = 21,760 - 1,360 = 20,400 ms. This is greater than the value of 20 seconds for the case where T(i) = 10, and so option 1 cannot meet PR3 in all cases.

However, in evaluating architectural option 1's response to PR3, we must ask "What is the cost of a missed update?" and "Is it ever acceptable to violate this requirement?" In some safety-critical applications, the answer would be "no." In most applications, the answer would be "yes," providing that this occurrence was infrequent. The results of this evaluation force one to reconsider the importance of meeting PR3.

## 8.2   Performance Analysis of Option 2

The performance characteristics of architectural option 2 are summarized in Table 8-2.

| WCCL | ACPL | Jitter |
|------|------|--------|
| 20,560 ms | 2,550 ms | 9,520 ms |

*Table 8-2:   Performance Analysis for Option 2*

One point (which will be discussed further later in this paper) should be noted here: if one of the servers fails, option 2 has the performance and availability characteristics of option 1.

## 8.3   Performance Analysis of Option 3

The performance characteristics of architectural option 3 are summarized in Table 8-3.

| WCCL | ACPL | Jitter |
|------|------|--------|
| 41,120 ms | 5,200 ms | ≤20,400 ms |

*Table 8-3:   Performance Analysis for Option 3*

For this analysis, we have added a new factor: servicing the intelligent cache (adding a new update and recalculating the extrapolation model) takes 100 ms. In this case, the worst- case

---

1.    WCPL = worst-case periodic latency; BCPL = best-case periodic latency

jitter is exactly the same as for option 1: 20,400 ms. *However*, the intelligent cache exists to protect the client against some amount of lost data. As a consequence, it can bound the worst-case jitter. When some preset time period elapses, the intelligent cache can pass a synthesized update to the client. When the actual update arrives, the cache updates its state accordingly. Thus, if *we trust the intelligent cache*, we can bound the worst-case jitter to any desired value. The smaller the bounding value, the more likely a given update will be synthesized by the intelligent cache rather than coming directly from the server.

## 8.4   Critique of the Analysis

This simple performance analysis gives insight into the characteristics of each solution early in the design process, as befits an *architectural-level* analysis. As more details are required, the analyses can be refined, using techniques such as Rate Monotonic Analysis (RMA) [Klein 93], SPE [Smith 93], simulation, or prototyping. More importantly, a high-level analysis guides our future investigations, highlighting potential performance "hot-spots," and allowing us to determine areas of architectural sensitivity to performance, which lead us to the location of tradeoff points.

The ATAM thus promotes analysis at multiple resolutions as a means of minimizing risk at acceptable levels of cost. Areas of high risk are analyzed more deeply (perhaps simulated or prototyped) than the rest of the architecture. And each level of analysis helps determine where to analyze more deeply in the next iteration.

# 9 Realize Scenarios/Availability Analyses

In the availability analysis that follows, we only present the results for the case of 24 failures per year. As specified by the availability scenarios, we consider two classes of repairs, depending on the type of failure:

- major failures, such as a burned-out power supply, that require a visit by a hardware technician to repair, taking 1/2 a day

- minor failures, such as software bugs, that can be "repaired" by rebooting the system, taking 10 minutes

To understand the availability of each of the architectural options, we built and solved a Markov model. In this analysis, we only considered server availability.

## 9.1 Availability Analysis of Option 1

Solving the Markov model for option 1 gives the results shown in Table 9-1: 279 hours of down time per year for the burned-out power supply and almost 4 hours down per year for the faulty operating system.

| Repair Time | Failures/yr | Availability | Hrs down/yr |
|-------------|-------------|--------------|-------------|
| 12 hours | 24. | 0.96817 | 278.833 |
| 10 minutes | 24. | 0.99954 | 3.9982 |

*Table 9-1: Availability of Option 1*

## 9.2 Availability Analysis of Option 2

We would expect option 2 to have better availability than option 1, since each server acts as a backup for the other, and we expect the probability of both servers being unavailable to be small. Solving the Markov model for this architecture, we get the results shown in Table 9-2.

| Repair Time | Failures/yr | Availability | Hrs down/yr |
|-------------|-------------|--------------|-------------|
| 12 hours | 24. | 0.99798 | 17.7327 |
| 10 minutes | 24. | ~1.0 | 0.0036496 |

*Table 9-2:  Availability of Option 2*

Table 9-2 shows that option 2 now suffers almost 18 hours of down time per year in the burned-out power supply case. This indicates that architectural option 2 might still suffer outages if it encounters frequent hardware problems. On the other hand, option 2 shows near-perfect availability in the operating system reboot scenario. The availability is shown as perfect 1.0 (the calculations were performed to 5 digits of accuracy). In the worst case of 24 annual failures option 2 exhibits only 13 *seconds* of down time per year.

## 9.3 Availability Analysis of Option 3

Considering architectural option 3, we expect that it will have better availability characteristics than option 1, but worse than option 2. This is because the intelligent cache, while providing some resistance to server failures, is not expected to be as trustworthy as an independent server. Solving the Markov model, we get the results shown in Table 9-3 for a cache that is assumed to be trustworthy for 5 minutes.

| Repair Time | Failures/yr | Reliability | Hrs down/yr |
|-------------|-------------|-------------|-------------|
| 12 hours | 24. | 0.96839 | 276.91 |
| 10 minutes | 24. | 0.9997 | 2.66545 |

*Table 9-3:  Availability of Option 3*

The results in Table 9-3 show that the 5-minute intelligent cache does little to improve option 3 over option 1 in the scenario with the burnt-out power supply. Option 3 still suffers over 277 hours of down time per year. However, the results for the reboot scenario look more encouraging. The cache reduces down time to 2.7 hours per year. Thus, it appears that the intelligent cache, if its extrapolation was improved, might provide high availability at low cost (since this option uses a single server, compared with the replicated servers used in option 2). We will return to this issue shortly.

# 10    Critique of the Options

Now that we have seen two different attribute analyses, we can comment on the level of granularity at which a system is analyzed. The ATAM advocates analysis at multiple levels of resolution as a means of minimizing risk at acceptable investments of time and effort. Areas that are deemed to be of high risk are analyzed and evaluated more deeply than the rest of the architecture. And each level of analysis helps to determine "hot spots" to focus on in the next iteration. We will illustrate this point next.

The three architectures can be partially characterized and understood by the measures that we have just derived. From this analysis, we can conclude the following:

- Option 1 has poor performance and availability. It is also the least expensive option (in terms of hardware costs; the detailed cost analyses can be found in [Barbacci 97]).

- Option 2 has excellent availability, but at the cost of extra hardware. It also has excellent performance (when both servers are functioning), and the characteristics of option 1 when a single server is down.

- Option 3 has slightly better availability than option 1, better performance than option 1 (in that the worst-case jitter can be bounded), slightly greater cost than option 1, and lower cost than option 2.

The conclusions of our analyses also cause us to ask some further questions.

## 10.1  Further Investigation of Option 2

For example, we need to consider the nature of option 2 with a server failure. Given that option 2 is identical to option 1 when one server fails, and we have already concluded that option 1 has poor performance and availability, it is important to know how much time option 2 will be in that reduced state of service. When we calculate the availability of *both* servers, using our worst-case assumption of 24 failures per year, we expect to suffer over 22 days of reduced service.

## 10.2  Action Plan

Given this understanding of options 2 and 3, we see that neither completely meets its requirements. While option 2 meets its availability target (for failures that involve rebooting the server), it leaves the system in a state where its performance targets can not be met for more

than 22 days per year. Perhaps a combination of options 2 and 3—dual servers and intelligent cache on clients—will be a better alternative. This option will provide the superior availability and performance of option 2, but during the times when one server has failed, we mitigate the jitter problems of the single remaining server by using the intelligent cache.

We could not have made these design decisions without knowledge gained from the analysis. Performing a multi-attribute analysis allows one to understand the strengths and weaknesses of a system, and of the parts of a system, within a framework that supports making design decisions.

# 11 Sensitivity Analyses

Given that the performance and availability of option 2 were so much better than option 1, we would suspect that these attributes are sensitive to the number of servers. Sensitivity analysis confirms this: performance increases linearly as the number of servers increases (up to the point where there is 1 server per client), and availability increases by roughly an order of magnitude with each additional server [Barbacci 97].

Given that option 3 has some desirable characteristics in terms of cost and jitter, we might ask if we can improve the intelligent cache sufficiently to make this option acceptable from an availability perspective. To answer this, we plot option 3's availability against the length of time during which the intelligent cache's data is trusted. This plot is shown in Table 11-1.
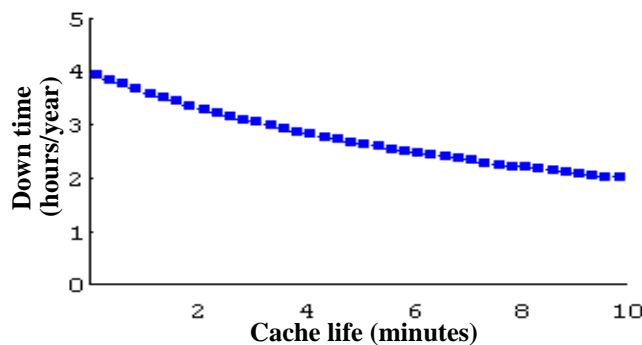


*Figure 11-1:    Down Time vs. Intelligent Cache Life*

As we can see, an improved intelligent cache does improve availability. However, the rate of improvement in availability as a function of cache life is so small that no reasonable, *achievable* amount of cache improvement will result in the kind of availability demonstrated for option 2. In effect, the intelligent cache is an architectural barrier with respect to availability, because it can not be made to achieve the levels of utility required of it. To put it another way, the availability of option 3 is not *sensitive* to cache life. To increase the availability substantially, other paths must be investigated.

# 12   Security Analyses

Although we could have been conducting security analyses with the performance and availability analyses from the start, the ATAM does not require that all attributes be analyzed in parallel. The ATAM allows the designer to focus on those attributes that are considered to be primary, and then introduce others later on. This can lead to cost benefits in applying the method, since what may be costly analyses for some secondary attributes need not be applied to architectures that were unsuitable for the primary attributes. Though all analyses need not occur "up front and simultaneously," the analyses for the secondary attributes can still occur well before implementation begins.

We will now do another iteration of the ATAM and analyze our three options in terms of their security. In particular, we will examine the connections between the furnace servers and clients, since this could be the subject of an attack. The object at risk is the temperature sent from the server to the client, since this information is used by the client to adjust the furnace settings. Having the temperature tampered with could be a significant safety concern. This leads us to the following security requirement:

   SR1: The temperature readings must not be corrupted before they arrive at the client.

From this generic requirement we will consider the following more specific security scenarios:

- A threat object between the server and operators modifies the temperatures received by the operators.
- A server is spoofed to misrepresent temperatures received by the operators.

Our initial security investigation of the architectural options must, once again, make some environmental assumptions. These assumptions are dependent on the operational environment of the delivered system and include factors such as operator training and patch management. These dependencies are out of scope for the analysis at this level of detail, but must be considered later in the design process.

So, to calculate the probability of a successful attack within an acceptable window of opportunity for an intruder, we define initial values that are *reasonable* for the functions provided in the RTS architectures. These values are shown in Table 12-1:

| Attack Components | | Value |
|---|---|---|
| Attack Exposure Window | | 60 minutes |
| Attack Rate | | 0.05 systems/min |
| Server failure rate | | 24 failures/year |
| Prob of server failure within 1 hour | | 0.0027 |
| Prob of successful | TCP Intercept | 0.5 |
| | Spoof IP address | 0.9 |
| | Kill Connection | 0.75 |
| | Kill Server | 0.25 |

*Table 12-1:   Environmental Security Assumptions*

In addition, we will posit two attack scenarios: one where the intruder uses a "man in the middle" (MIM) attack, and one where the intruder uses a "spoof-server" attack.

For the MIM attack, the attacker uses a TCP intercept tool to modify the values of the temperatures during transmission. Since there are no specific security countermeasures to this attack, the only barrier is the 60-minute window of opportunity and the 0.5 probability of success for the TCP intercept tool. Thus, the rate of successful attack is 0.025 systems/minute, or about 1.5 successful attacks expected in the window of opportunity.

For the spoof-server attack, there are three possible ways to succeed. The intruder could wait for a server to fail, then spoof that server's address and take over the client connections. This presumes that the intruder can determine when a server has failed and take advantage of this before the clients time out. Another successful method would be to cause the server to fail (the "kill-server" attack), then take over the connections. A third is to disrupt the connections between the client and server, then establish new connections as a spoofed server (the "kill-connection" attack). For this analysis, it is presumed that the intruder is equally likely to attempt any of these methods in a given attack, and the results are summarized in Table 12-2. Of course, these numbers appear precise, but must be treated as estimates given the subjective nature of the environmental assumptions.

| Attack Type | Expected Intrusions in 60 Mins |
|---|---|
| Kill Connection | 2.04 |
| Kill Server | 0.66 |
| Server Failure | 0.0072 |

*Table 12-2:   Anticipated Spoof-Attack Success Rates*

It should be noted that if the system must deal with switching servers and reconnecting clients when a server goes in and out of service, it will be easier for an intruder to spoof a server and convince a client to switch to the bogus server. We will return to this point in the sensitivity analysis.

The results of this analysis show that in each case, it is expected that a penetration will take place within 60 minutes. For the MIM scenario, the expected number of successful attacks is 1.5, indicating that an intruder would have more than enough time to complete the attack before detection. For the spoof attack, the number of successful attacks ranges from 0.0072 to just over 2, again showing that a penetration using this technique is also likely.

## 12.1  Refined Architectural Options

To address the apparent inadequacy of the three options, we need to cycle around the steps of the ATAM, proposing new architectures. The modified versions of the options include the addition of encryption/decryption and the use of the intelligent cache as an intrusion detection mechanism, as shown in Figure 12-1.



*Figure 12-1:    Security Modifications*

Encryption/decryption needs little explanation; it is the most common security "bolt-on" solution. The other security enhancement is not a topological change, but rather a change in the function of the intelligent cache. In this design, the cache uses its predictive values to determine if the temperatures supplied by the network are reasonable. A temperature that is significantly outside a reasonable change range is deemed to be generated by an intruder, and thus

the cache aids the operator in detecting an intrusion. Adding encryption adds some new environmental assumptions which are listed in Table 12-3:

| Attack Components | | Value |
|---|---|---|
| Prob of successful | Decrypt | 0.0005 |
| | Replay | 0.05 |
| | Key Distribution | 0.09 |

*Table 12-3:  Additional Security Assumptions*

Based on these assumptions, we can calculate the expected number of intrusions. Not surprisingly, the addition of encryption has reduced these substantially—by at least an order of magnitude—in each option, as shown in Table 12-4:

| Attack Type | Expected Intrusions in 60 Mins |
|---|---|
| Kill Connection | 0.18225 |
| Kill Server | 0.03375 |
| Server Failure | 0.0006 |

*Table 12-4:  Spoof Success Rates with Encryption*

Our analysis of the intelligent cache changes only one environmental assumption: the "Attack Exposure Window" goes down to 5 minutes, since we assume that an operator can detect and respond to an intrusion in that time. Using this form of intrusion detection reduces the number of expected intrusions by 1-2 orders of magnitude, giving a result comparable to encryption, but at substantially lower performance and software/hardware costs, as shown in Table 12-5:

| Attack Type | Expected Intrusions in 60 Mins |
|---|---|
| Kill Connection | 0.16875 |
| Kill Server | 0.05625 |
| Server Failure | 0.005 |

*Table 12-5:  Spoof Success Rates with Intrusion Detection*

At this point, new performance and availability analyses will need to be run to account for the additional functionality and hardware required by the intelligent cache or encryption modifications, thus instigating another trip around the spiral.

# 13   Sensitivities and Tradeoffs

Following the ATAM, we are now in a position to do further sensitivity analysis. In particular, we noted earlier that both availability and performance correlated positively to the number of servers. A sensitivity analysis with respect to security shows just the opposite—security is negatively correlated with the number of servers. This is for two reasons:

- Going from one to multiple servers requires additional logic within the clients, so that they are able to switch between servers. This provides an entry point for spoofing attacks that does not exist when a client is "hard-wired" to a single server.

- The probability of a server failure within one hour increases linearly with the number of servers, thus increasing the opportunities for server spoofing.

At this point, we have discovered an *architectural* tradeoff point in the number of servers. Performance and availability are correlated positively, while security and presumably cost are correlated negatively with the number of servers. We cannot maximize cost, performance, availability, and security simultaneously. Using this information, we can make informed tradeoff decisions regarding the level of the various attributes that we can achieve at an acceptable cost, and we can do so within an analytic framework.

# 14   The Implications of the ATAM

For every assumption that we make in a system's design, we trade cost for knowledge. For example, if a periodic update is supposed to arrive every 10 seconds, do we want it to arrive exactly every 10 seconds, on average every 10 seconds, some time within each 10-second window? To give another example, consider the requirement detailing the worst-case latency of control packets. As discussed earlier, is this worst case *ever* acceptable? If so, how frequently can we tolerate it? The process of analyzing architectural attributes forces us to try to answer these questions. Either we understand our requirements precisely or we pay for ignorance by overengineering or underengineering the system. If we overengineer, we pay for our ignorance by making the system needlessly expensive. If we underengineer, we face system failures, losing customers or perhaps even lives.

We cannot believe the numbers that we generated in our analyses. However we can believe the *trends*—we see order of magnitude differences among designs—and these differences, along with sensitivity analysis, tell us where to investigate further, where to get better environmental information, and where to prototype, which will get us numbers that we can believe. Every analysis step that we take precipitates new questions. While this seems like a daunting, never-ending prospect, it is manageable because these questions are posed and answered within an analytic attribute framework, and because in architectural analysis we are more interested in finding large effects than in precise estimates.

In addition to concretizing requirements, the ATAM has one other benefit: it helps to uncover implicit requirements. This occurs because attribute analyses are, as we have seen, *interdependent*—they depend, at least partially, on a common set of elements, such as the number of servers. However, in the past, they have been modeled as though they were *independent*. This is clearly not the case.

Each analyzed attribute has implications for other attributes. For example, although the availability analysis was only focused on server availability, in a complete analysis, we would look at potential failures of all components, and we would look at different failure types such as dropping a message. If the communication channel is not reliable, we might want to plan for re-sending messages. To do this involves additional computation. Thus one of the implications of this availability concern is that the *performance* models of the options under consideration need to be modified. To recap, we discover attribute interactions in two ways: using sensitivity analysis to find tradeoff points, and by examining the assumptions that we make for analysis *A* while performing analysis *B*. The "no re-sending messages" assumption is one example of an

interaction. This assumption, if false, may have implications for safety, and availability. A solution will have implications for performance.

In the ATAM, attribute experts create and analyze their models independently and then exchange information (clarifying or creating new requirements). On the basis of this information, they refine their models. The *interaction* of their analyses and the identification of tradeoffs has a greater effect on system understanding and stakeholder communication than any of those analyses could do on their own.

The complexity inherent in most real-world software design implies that an architecture tradeoff analysis will rarely be a straightforward activity that allows you to proceed linearly to a perfect solution. Each step of the method answers some design questions and brings some issues into sharper focus. However, each step often raises new questions and reveals new interactions between attributes which may require further analysis, sometimes at different levels of abstraction. Such obstacles are an intrinsic part of a detailed methodical exploration of the design space and cannot be avoided. Managing the conflicts and interactions that are revealed by the ATAM places heavy demands on the analysis skills of the individual attribute experts. Success depends largely on the ability of those experts to transcend barriers of differing terminology and methodology to understand the implications of inter-attribute dependencies, and to jointly devise candidate architectural solutions for further analysis. As burdensome as this may appear to be, it is better to intensively manage these attribute interactions early in the design process than to wait until some unfortunate consequences of the interactions are revealed in a deployed system.

# 15 Conclusions

The ATAM was motivated by a desire to make rational choices among competing architectures, based upon well-documented analyses of system attributes, concentrating on the identification of tradeoff points. The ATAM also serves as a vehicle for the early clarification of requirements. As a result of performing an architecture tradeoff analysis, we have an enhanced understanding of, and confidence in, a system's ability to meet its requirements. We also have a documented rationale for the architectural choices made, consisting of both the scenarios used to motivate the attribute-specific analyses and the results of those analyses.

Consider the RTS case study: we began with vague requirements and enumerated three architectural options. The analytical framework helped determine the useful characteristics of each of the architectural options and highlighted the costs and benefits of the architectural features. More importantly, the ATAM helped determine the locations of architectural tradeoff points, which helped us understand the limits of each option. This helped us develop informed action plans for modifying the architecture, leading to new evaluations and new iterations of the method.

# References

**[Barbacci 97]**      Barbacci, M.; Carriere, J.; Kazman, R.; Klein, M.; Lipson, H.; Longstaff, T.; Weinstock, C. *Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis.* (CMU/SEI-97-TR-29 ADA188100). Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute, 1997.

**[Bass 98]**      Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*, Reading, MA: Addison-Wesley, 1998.

**[Boehm 86]**      Boehm, B. "A Spiral Model of Software Development and Enhancement." *ACM Software Eng. Notes 11,* 4 (August 1986): 22-42.

**[Kazman 96]**      Kazman, R.; Abowd, G.; Bass, L.; & Clements, P. "Scenario-Based Analysis of Software Architecture." *IEEE Software 13,* 6 (November 1996): 47-55.

**[Klein 93]**      Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & Gonzales Harbour, M. *A Practitioner's Handbook for Real-Time Analysis.* Boston, MA: Kluwer Academic, 1993.

**[McCall 94]**      McCall, J. "Quality Factors", 958-969. *Encyclopedia of Software Engineering* (Marciniak, J., ed.). Vol. 2. New York, NY: Wiley, 1994.

**[Smith 93]**      Smith, C. & Williams, L. "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives." *IEEE Transactions on Software Engineering 19,* 7 (July 1993): 720-741.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

| 1. **AGENCY USE ONLY** (leave blank) | 2. **REPORT DATE** <br> July 1998 | 3. **REPORT TYPE AND DATES COVERED** <br> Final |
|---|---|---|

| 4. **TITLE AND SUBTITLE** <br> The Architecture Tradeoff Analysis Method | 5. **FUNDING NUMBERS** <br> C — F19628-95-C-0003 |
|---|---|
| 6. **AUTHOR(S)** <br> Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, Jeromy Carriere | |

| 7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** <br> Software Engineering Institute <br> Carnegie Mellon University <br> Pittsburgh, PA 15213 | 8. **PERFORMING ORGANIZATION REPORT NUMBER** <br> CMU/SEI-98-TR-008 |
|---|---|

| 9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** <br> HQ ESC/DIB <br> 5 Eglin Street <br> Hanscom AFB, MA 01731-2116 | 10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** <br> ESC-TR-98-008 |
|---|---|

11. **SUPPLEMENTARY NOTES**

| 12.a **DISTRIBUTION/AVAILABILITY STATEMENT** <br> Unclassified/Unlimited, DTIC, NTIS | 12.b **DISTRIBUTION CODE** |
|---|---|

13. **ABSTRACT** (maximum 200 words)

This paper presents the Architecture Tradeoff Analysis Method (ATAM), a structured technique for understanding the tradeoffs inherent in the architectures of software-intensive systems. This method was developed to provide a principled way to evaluate a software architecture's fitness with respect to multiple competing quality attributes: modifiability, security, performance, availability, and so forth. These attributes interact, and improving one often comes at the price of worsening one or more of the others. The method helps us reason about architectural decisions that affect quality attribute interactions. The ATAM is a spiral model of design, one of postulating candidate architectures followed by analysis and risk mitigation that lead to refined architectures.

| 14. **SUBJECT TERMS** <br> architecture analysis, software architecture, tradeoffs | 15. **NUMBER OF PAGES** <br> 40 |
|---|---|
| | 16. **PRICE CODE** |

| 17. **SECURITY CLASSIFICATION OF REPORT** <br> UNCLASSIFIED | 18. **SECURITY CLASSIFICATION OF THIS PAGE** <br> UNCLASSIFIED | 19. **SECURITY CLASSIFICATION OF ABSTRACT** <br> UNCLASSIFIED | 20. **LIMITATION OF ABSTRACT** <br> UL |
|---|---|---|---|

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102