Part Two: Creating an Architecture

Part One of this book introduced the Architecture Business Cycle (ABC) and laid the groundwork for the study of software architecture. In particular, it set out the influences at work when an architect begins building a system, and it pointed out that requirements for particular quality attributes such as performance or modifiability often originate from the organization's business goals. How then does an architect create an architecture? That is the focus of Part Two. Because the achievement of quality attributes is critical to the success of a system, we begin by discussing quality and how it is achieved with the contents of the architect's tool box.

Quality is often in the eye of the beholder (to paraphrase Booth Tarkington). What this means for the architect is that customers may dislike a design because their concept of quality differs from the architect's. Quality attribute scenarios are the means by which quality moves from the eye of the beholder to a more objective basis. In Chapter 4, we explore different types of quality that may be appropriate for an architecture. For six important attributes (availability, modifiability, performance, security, testability, and usability), we describe how to generate scenarios that can be used to characterize quality requirements. These scenarios demonstrate what quality means for a particular system, giving both the architect and the customer a basis for judging a design.

Knowing the quality requirements, of course, only provides a goal for the architect. In Chapter 5, we list the tools (tactics and patterns) in the architect's kit that are used to achieve the quality attributes. High availability, for example, depends on having some form of redundancy in either data or code, and this redundancy generates additional considerations for the architect (such as ensuring synchronization among the replicates).

In Chapter 6, we introduce our second case study—a system designed to support the air traffic control functions of the Federal Aviation Administration. This system was designed to achieve ultra-high availability requirements (less than five minutes downtime per year) and illustrates the tactics enumerated in Chapter 5.

Quality attribute scenarios and architectural tactics are some of the tools available for the creation of an architecture. In Chapter 7, we discuss how to apply these tools in designing an architecture and in building a skeletal system, and how the architecture is reflected in the organizational structure.

In Chapter 8, we present our third case study, of flight simulators. These systems were designed to achieve real-time performance and to be readily modified. We show how these goals were achieved.

Once an architecture has been designed, it must be documented. This is a matter of documenting first the relevant views and then the material that extends beyond any particular view. Chapter 9 details how to document an architecture.

Frequently, the architecture for a system is unavailable—because it was never documented, it has been lost, or the as-built system differs from the designed system. Chapter 10 discusses recovering the architecture for an existing system.

Chapter 4. Understanding Quality Attributes

with Felix Bachmann and Mark Klein

Note: Felix Bachmann and Mark Klein are senior members of the technical staff at the Software Engineering Institute.

"Cheshire-Puss," [Alice] began, rather timidly … "Would you tell me, please, which way I ought to go from here?" "That depends a good deal on where you want to go to," said the Cat. "Oh, I don't much care where—" said Alice. Then it doesn't matter which way you go," said the Cat. "—so long as I get somewhere," said Alice. "Oh, you're sure to do that," said the Cat, "if only you walk long enough."

—Lewis Carroll, Alice's Adventures in Wonderland.

As we have seen in the Architecture Business Cycle, business considerations determine qualities that must be accommodated in a system's architecture. These qualities are over and above that of functionality, which is the basic statement of the system's capabilities, services, and behavior. Although functionality and other qualities are closely related, as you will see, functionality often takes not only the front seat in the development scheme but the only seat. This is short-sighted, however. Systems are frequently redesigned not because they are functionally deficient—the replacements are often functionally identical—but because they are difficult to maintain, port, or scale, or are too slow, or have been compromised by network hackers. In Chapter 2, we said that architecture was the first stage in software creation in which quality requirements could be addressed. It is the mapping of a system's functionality onto software structures that determines the architecture's support for qualities. In Chapter 5 we discuss how the qualities are supported by architectural design decisions, and in Chapter 7 we discuss how the architect can manage the tradeoffs inherent in any design.

Here our focus is on understanding how to express the qualities we want our architecture to provide to the system or systems we are building from it. We begin the discussion of the relationship between quality attributes and software architecture by looking closely at quality attributes. What does it mean to say that a system is modifiable or reliable or secure? This chapter characterizes such attributes and discusses how this characterization can be used to express the quality requirements for a system.

4.1. Functionality and Architecture

Functionality and quality attributes are orthogonal. This statement sounds rather bold at first, but when you think about it you realize that it cannot be otherwise. If functionality and quality attributes were not orthogonal, the choice of function would dictate the level of security or performance or availability or usability. Clearly though, it is possible to independently choose a desired level of each. Now, this is not to say that any level of any quality attribute is achievable with any function. Manipulating complex graphical images or sorting an enormous database might be inherently complex, making lightning-fast performance impossible. But what is possible is that, for any of these functions your choices as an architect will determine the relative level of quality. Some architectural choices will lead to higher performance; some will lead in the other direction. Given this understanding, the purpose of this chapter is, as with a good architecture, to separate concerns. We will examine each important quality attribute in turn and learn how to think about it in a disciplined way.

What is functionality? It is the ability of the system to do the work for which it was intended. A task requires that many or most of the system's elements work in a coordinated manner to complete the job, just as framers, electricians, plumbers, drywall hangers, painters, and finish carpenters all come together to cooperatively build a house. Therefore, if the elements have not been assigned the correct responsibilities or have not been endowed with the correct facilities for coordinating with other elements (so that, for instance, they know when it is time for them to begin their portion of the task), the system will be unable to offer the required functionality.

Functionality may be achieved through the use of any of a number of possible structures. In fact, if functionality were the only requirement, the system could exist as a single monolithic module with no internal structure at all. Instead, it is decomposed into modules to make it understandable and to support a variety of other purposes. In this way, functionality is largely independent of structure. Software architecture constrains its allocation to structure when other quality attributes are important. For example, systems are frequently divided so that several people can cooperatively build them (which is, among other things, a time-to-market issue, though seldom stated this way). The interest of functionality is how it interacts with, and constrains, those other qualities.

4.2. Architecture and Quality Attributes

Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct. For example:

- Usability involves both architectural and nonarchitectural aspects. The nonarchitectural aspects include making the user interface clear and easy to use. Should you provide a radio button or a check box? What screen layout is most intuitive? What typeface is most clear? Although these details matter tremendously to the end user and influence usability, they are not architectural because they belong to the details of design. Whether a system provides the user with the ability to cancel operations, to undo operations, or to re-use data previously entered is architectural, however. These requirements involve the cooperation of multiple elements.
- Modifiability is determined by how functionality is divided (architectural) and by coding techniques within a module (nonarchitectural). Thus, a system is modifiable if changes involve the fewest possible number of distinct elements. This was the basis of the A-7E module decomposition structure in Chapter 3. In spite of having the ideal architecture, however, it is always possible to make a system difficult to modify by writing obscure code.
- Performance involves both architectural and nonarchitectural dependencies. It depends partially on how much communication is necessary among components (architectural), partially on what functionality has been allocated to each component (architectural), partially on how shared resources are allocated (architectural), partially on the choice of algorithms to implement selected functionality (nonarchitectural), and partially on how these algorithms are coded (nonarchitectural).

The message of this section is twofold:

1. Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and can be evaluated at the architectural level.
2. Architecture, by itself, is unable to achieve qualities. It provides the foundation for achieving quality, but this foundation will be to no avail if attention is not paid to the details.

Within complex systems, quality attributes can never be achieved in isolation. The achievement of any one will have an effect, sometimes positive and sometimes negative, on the achievement of others. For example, security and reliability often exist in a state of mutual tension: The most secure system has the fewest points of failure—typically a security kernel. The most reliable system has the most points of failure—typically a set of redundant processes or processors where the failure of any one will not cause the system to fail. Another example of the tension between quality attributes is that almost every quality attribute negatively affects performance. Take portability. The main technique for achieving portable software is to isolate system dependencies, which introduces overhead into the system's execution, typically as process or procedure boundaries, and this hurts performance.

Let's begin our tour of quality attributes. We will examine the following three classes:

1. Qualities of the system. We will focus on availability, modifiability, performance, security, testability, and usability.
2. Business qualities (such as time to market) that are affected by the architecture.
3. Qualities, such as conceptual integrity, that are about the architecture itself although they indirectly affect other qualities, such as modifiability.

4.3. System Quality Attributes

System quality attributes have been of interest to the software community at least since the 1970s. There are a variety of published taxonomies and definitions, and many of them have their own research and practitioner communities. From an architect's perspective, there are three problems with previous discussions of system quality attributes:

- The definitions provided for an attribute are not operational. It is meaningless to say that a system will be modifiable. Every system is modifiable with respect to one set of changes and not modifiable with respect to another. The other attributes are similar.
- A focus of discussion is often on which quality a particular aspect belongs to. Is a system failure an aspect of availability, an aspect of security, or an aspect of usability? All three attribute communities would claim ownership of a system failure.
- Each attribute community has developed its own vocabulary. The performance community has "events" arriving at a system, the security community has "attacks" arriving at a system, the availability community has "failures" of a system, and the usability community has "user input." All of these may actually refer to the same occurrence, but are described using different terms.

A solution to the first two of these problems (nonoperational definitions and overlapping attribute concerns) is to use quality attribute scenarios as a means of characterizing quality attributes. A solution to the third problem is to provide a brief discussion of each attribute—concentrating on its underlying concerns—to illustrate the concepts that are fundamental to that attribute community.

QUALITY ATTRIBUTE SCENARIOS

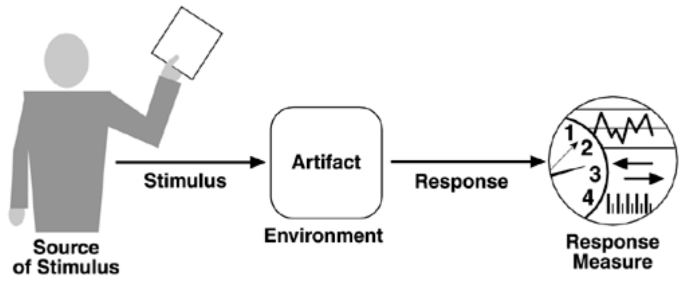
A quality attribute scenario is a quality-attribute-specific requirement. It consists of six parts.

- Source of stimulus. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
- Stimulus. The stimulus is a condition that needs to be considered when it arrives at a system.
- Environment. The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
- Artifact. Some artifact is stimulated. This may be the whole system or some pieces of it.
- Response. The response is the activity undertaken after the arrival of the stimulus.
- Response measure. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

We distinguish general quality attribute scenarios (general scenarios)—those that are system independent and can, potentially, pertain to any system—from concrete quality attribute scenarios (concrete scenarios)—those that are specific to the particular system under consideration. We present attribute characterizations as a collection of general scenarios; however, to translate the attribute characterization into requirements for a particular system, the relevant general scenarios need to be made system specific.

Figure 4.1 shows the parts of a quality attribute scenario.

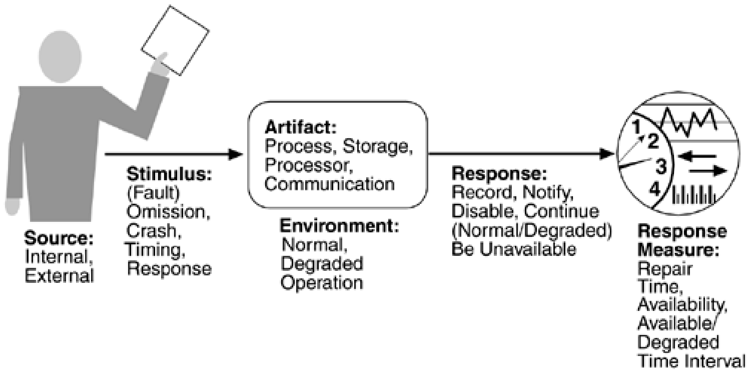
Figure 4.1. Quality attribute parts



Availability Scenario

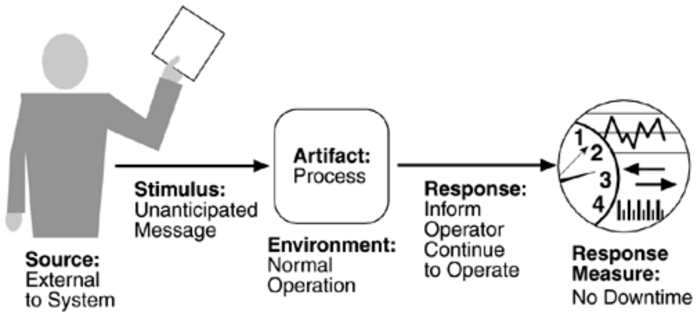
A general scenario for the quality attribute of availability, for example, is shown in Figure 4.2. Its six parts are shown, indicating the range of values they can take. From this we can derive concrete, system-specific, scenarios. Not every system-specific scenario has all of the six parts. The parts that are necessary are the result of the application of the scenario and the types of testing that will be performed to determine whether the scenario has been achieved.

Figure 4.2. Availability general scenarios


Source: Internal, External
Stimulus: (Fault) Omission, Crash, Timing, Response
Artifact: Process, Storage, Processor, Communication
Environment: Normal, Degraded Operation
Response: Record, Notify, Disable, Continue (Normal/Degraded) Be Unavailable
Response Measure: Repair Time, Availability, Available/ Degraded Time Interval
1
2
3
4

An example availability scenario, derived from the general scenario of Figure 4.2 by instantiating each of the parts, is "An unanticipated external message is received by a process during normal operation. The process informs the operator of the receipt of the message and continues to operate with no downtime." Figure 4.3 shows the pieces of this derived scenario.

Figure 4.3. Sample availability scenario


Source: External to System
Stimulus: Unanticipated Message
Artifact: Process
Environment: Normal Operation
Response: Inform Operator Continue to Operate
Response Measure: No Downtime
1
2
3
4

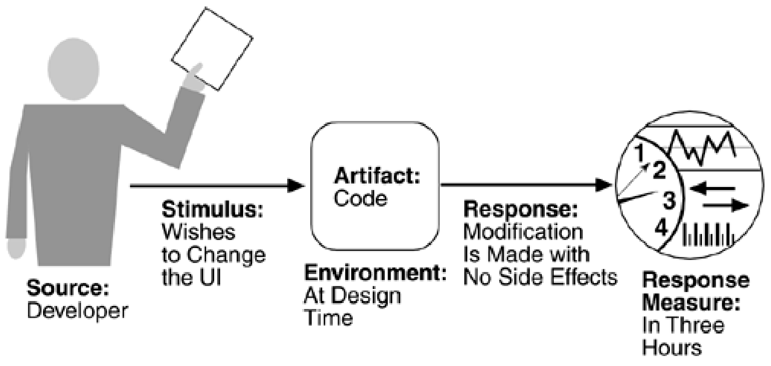
The source of the stimulus is important since differing responses may be required depending on what it is. For example, a request from a trusted source may be treated differently from a request from an untrusted source in a security scenario. The environment may also affect the response, in that an event arriving at a system may be treated differently if the system is already overloaded. The artifact that is stimulated is less important as a requirement. It is almost always the system, and we explicitly call it out for two reasons.

First, many requirements make assumptions about the internals of the system (e.g., "a Web server within the system fails"). Second, when we utilize scenarios within an evaluation or design method, we refine the scenario artifact to be quite explicit about the portion of the system being stimulated. Finally, being explicit about the value of the response is important so that quality attribute requirements are made explicit. Thus, we include the response measure as a portion of the scenario.

Modifiability Scenario

A sample modifiability scenario is "A developer wishes to change the user interface to make a screen's background color blue. This change will be made to the code at design time. It will take less than three hours to make and test the change and no side effect changes will occur in the behavior." Figure 4.4 illustrates this sample scenario (omitting a few minor details for brevity).

Figure 4.4. Sample modifiability scenario



Source: Developer
Stimulus: Wishes to Change the UI
Artifact: Code
Environment: At Design Time
Response: Modification Is Made with No Side Effects
Response Measure: In Three Hours
1
2
3
4

A collection of concrete scenarios can be used as the quality attribute requirements for a system. Each scenario is concrete enough to be meaningful to the architect, and the details of the response are meaningful enough so that it is possible to test whether the system has achieved the response. When eliciting requirements, we typically organize our discussion of general scenarios by quality attributes; if the same scenario is generated by two different attributes, one can be eliminated.

For each attribute we present a table that gives possible system-independent values for each of the six parts of a quality scenario. A general quality scenario is generated by choosing one value for each element; a concrete scenario is generated as part of the requirements elicitation by choosing one or more entries from each column of the table and then making the result readable. For example, the scenario shown in Figure 4.4 is generated from the modifiability scenario given in Table 4.2 (on page 83), but the individual parts were edited slightly to make them read more smoothly as a scenario.

Concrete scenarios play the same role in the specification of quality attribute requirements that use cases play in the specification of functional requirements.

QUALITY ATTRIBUTE SCENARIO GENERATION

Our concern in this chapter is helping the architect generate meaningful quality attribute requirements for a system. In theory this is done in a project's requirements elicitation, but in practice this is seldom rigorously enforced. As we said in Chapter 1, a system's quality attribute requirements are seldom elicited and recorded in a disciplined way. We remedy this situation by generating concrete quality attribute scenarios. To do this, we use the quality-attribute-specific tables to create general scenarios and from these derive system-specific scenarios. Typically, not all of the possible general scenarios are created. The tables serve as a checklist to ensure that all possibilities have been considered rather than as an explicit generation mechanism. We are unconcerned about generating scenarios that do not fit a narrow definition of an attribute—if two attributes allow the generation of the same quality attribute requirement, the redundancy is easily corrected. However, if an important quality attribute requirement is omitted, the consequences may be more serious.

4.4. Quality Attribute Scenarios in Practice

General scenarios provide a framework for generating a large number of generic, system-independent, quality-attribute-specific scenarios. Each is potentially but not necessarily relevant to the system you are concerned with. To make the general scenarios useful for a particular system, you must make them system specific.

Making a general scenario system specific means translating it into concrete terms for the particular system. Thus, a general scenario is "A request arrives for a change in functionality, and the change must be made at a particular time within the development process within a specified period." A system-specific version might be "A request arrives to add support for a new browser to a Web-based system, and the change must be made within two weeks." Furthermore, a single general scenario may have many system-specific versions. The same system that has to support a new browser may also have to support a new media type.

We now discuss the six most common and important system quality attributes, with the twin goals of identifying the concepts used by the attribute community and providing a way to generate general scenarios for that attribute.

AVAILABILITY

Availability is concerned with system failure and its associated consequences. A system failure occurs when the system no longer delivers a service consistent with its specification. Such a failure is observable by the system's users—either humans or other systems. An example of an availability general scenario appeared in Figure 4.3.

Among the areas of concern are how system failure is detected, how frequently system failure may occur, what happens when a failure occurs, how long a system is allowed to be out of operation, when failures may occur safely, how failures can be prevented, and what kinds of notifications are required when a failure occurs.

We need to differentiate between failures and faults. A fault may become a failure if not corrected or masked. That is, a failure is observable by the system's user and a fault is not. When a fault does become observable, it becomes a failure. For example, a fault can be choosing the wrong algorithm for a computation, resulting in a miscalculation that causes the system to fail.

Once a system fails, an important related concept becomes the time it takes to repair it. Since a system failure is observable by users, the time to repair is the time until the failure is no longer observable. This may be a brief delay in the response time or it may be the time it takes someone to fly to a remote location in the mountains of Peru to repair a piece of mining machinery (this example was given by a person who was responsible for repairing the software in a mining machine engine.).

The distinction between faults and failures allows discussion of automatic repair strategies. That is, if code containing a fault is executed but the system is able to recover from the fault without it being observable, there is no failure.

The availability of a system is the probability that it will be operational when it is needed. This is typically defined as

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

From this come terms like 99.9% availability, or a 0.1% probability that the system will not be operational when needed.

Scheduled downtimes (i.e., out of service) are not usually considered when calculating availability, since the system is "not needed" by definition. This leads to situations where the system is down and users are waiting for it, but the downtime is scheduled and so is not counted against any availability requirements.

Availability General Scenarios

From these considerations we can see the portions of an availability scenario, shown in Figure 4.2.

- Source of stimulus. We differentiate between internal and external indications of faults or failure since the desired system response may be different. In our example, the unexpected message arrives from outside the system.
- Stimulus. A fault of one of the following classes occurs.

  - omission. A component fails to respond to an input.

  - crash. The component repeatedly suffers omission faults.

  - timing. A component responds but the response is early or late.

  - response. A component responds with an incorrect value.

  In Figure 4.3, the stimulus is that an unanticipated message arrives. This is an example of a timing fault. The component that generated the message did so at a different time than expected.

- Artifact. This specifies the resource that is required to be highly available, such as a processor, communication channel, process, or storage.
- Environment. The state of the system when the fault or failure occurs may also affect the desired system response. For example, if the system has already seen some faults and is operating in other than normal mode, it may be desirable to shut it down totally. However, if this is the first fault observed, some degradation of response time or function may be preferred. In our example, the system is operating normally.
- Response. There are a number of possible reactions to a system failure. These include logging the failure, notifying selected users or other systems, switching to a degraded mode with either less capacity or less function, shutting down external systems, or becoming unavailable during repair. In our example, the system should notify the operator of the unexpected message and continue to operate normally.
- Response measure. The response measure can specify an availability percentage, or it can specify a time to repair, times during which the system must be available, or the duration for which the system must be available. In Figure 4.3, there is no downtime as a result of the unexpected message.

Table 4.1 presents the possible values for each portion of an availability scenario.

Table 4.1. Availability General Scenario Generation

Portion of Scenario	**Possible Values**
Source	Internal to the system; external to the system
Stimulus	Fault: omission, crash, timing, response
Artifact	System's processors, communication channels, persistent storage, processes
Environment	Normal operation; degraded mode (i.e., fewer features, a fall back solution)
Response	System should detect event and do one or more of the following: record it notify appropriate parties, including the user and other systems disable sources of events that cause fault or failure according to defined rules be unavailable for a prespecified interval, where interval depends on criticality of system continue to operate in normal or degraded mode
Response Measure	Time interval when the system must be available Availability time Time interval in which system can be in degraded mode Repair time

MODIFIABILITY

Modifiability is about the cost of change. It brings up two concerns.

1. What can change (the artifact)? A change can occur to any aspect of a system, most commonly the functions that the system computes, the platform the system exists on (the hardware, operating system, middleware, etc.), the environment within which the system operates (the systems with which it must interoperate, the protocols it uses to communicate with the rest of the world, etc.), the qualities the system exhibits (its performance, its reliability, and even its future modifications), and its capacity (number of users supported, number of simultaneous operations, etc.). Some portions of the system, such as the user interface or the platform, are sufficiently distinguished and subject to change that we consider them separately. The category of platform changes is also called portability. Those changes may be to add, delete, or modify any one of these aspects.
2. When is the change made and who makes it (the environment)? Most commonly in the past, a change was made to source code. That is, a developer had to make the change, which was tested and then deployed in a new release. Now, however, the question of when a change is made is intertwined with the question of who makes it. An end user changing the screen saver is clearly making a change to one of the aspects of the system. Equally clear, it is not in the same category as changing the system so that it can be used over the Web rather than on a single machine. Changes can be made to the implementation (by modifying the source code), during compile (using compile-time switches), during build (by choice of libraries), during configuration setup (by a range of techniques, including parameter setting) or during execution (by parameter setting). A change can also be made by a developer, an end user, or a system administrator.

Once a change has been specified, the new implementation must be designed, implemented, tested, and deployed. All of these actions take time and money, both of which can be measured.

Modifiability General Scenarios

From these considerations we can see the portions of the modifiability general scenarios. Figure 4.4 gives an example: "A developer wishes to change the user interface. This change will be made to the code at design time, it will take less than three hours to make and test the change, and no side-effect changes will occur in the behavior."

- Source of stimulus. This portion specifies who makes the changes—the developer, a system administrator, or an end user. Clearly, there must be machinery in place to allow the system administrator or end user to modify a system, but this is a common occurrence. In Figure 4.4, the modification is to be made by the developer.
- Stimulus. This portion specifies the changes to be made. A change can be the addition of a function, the modification of an existing function, or the deletion of a function. It can also be made to the qualities of the system—making it more responsive, increasing its availability, and so forth. The capacity of the system may also change. Increasing the number of simultaneous users is a frequent requirement. In our example, the stimulus is a request to make a modification, which can be to the function, quality, or capacity.

  Variation is a concept associated with software product lines (see Chapter 14). When considering variation, a factor is the number of times a given variation must be specified. One that must be made frequently will impose a more stringent requirement on the response measures than one that is made only sporadically.

- Artifact. This portion specifies what is to be changed—the functionality of a system, its platform, its user interface, its environment, or another system with which it interoperates. In Figure 4.4, the modification is to the user interface.
- Environment. This portion specifies when the change can be made—design time, compile time, build time, initiation time, or runtime. In our example, the modification is to occur at design time.
- Response. Whoever makes the change must understand how to make it, and then make it, test it and deploy it. In our example, the modification is made with no side effects.
- Response measure. All of the possible responses take time and cost money, and so time and cost are the most desirable measures. Time is not always possible to predict, however, and so less ideal measures are frequently used, such as the extent of the change (number of modules affected). In our example, the time to perform the modification should be less than three hours.

Table 4.2 presents the possible values for each portion of a modifiability scenario.

Table 4.2. Modifiability General Scenario Generation

Portion of Scenario	**Possible Values**
Source	End user, developer, system administrator
Stimulus	Wishes to add/delete/modify/vary functionality, quality attribute, capacity
Artifact	System user interface, platform, environment; system that interoperates with target system
Environment	At runtime, compile time, build time, design time
Response	Locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification
Response	Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or

Table 4.2. Modifiability General Scenario Generation

Portion of Scenario	**Possible Values**
Measure	quality attributes

PERFORMANCE

Performance is about timing. Events (interrupts, messages, requests from users, or the passage of time) occur, and the system must respond to them. There are a variety of characterizations of event arrival and the response but basically performance is concerned with how long it takes the system to respond when an event occurs.

One of the things that make performance complicated is the number of event sources and arrival patterns. Events can arrive from user requests, from other systems, or from within the system. A Web-based financial services system gets events from its users (possibly numbering in the tens or hundreds of thousands). An engine control system gets its requests from the passage of time and must control both the firing of the ignition when a cylinder is in the correct position and the mixture of the fuel to maximize power and minimize pollution.

For the Web-based financial system, the response might be the number of transactions that can be processed in a minute. For the engine control system, the response might be the variation in the firing time. In each case, the pattern of events arriving and the pattern of responses can be characterized, and this characterization forms the language with which to construct general performance scenarios.

A performance scenario begins with a request for some service arriving at the system. Satisfying the request requires resources to be consumed. While this is happening the system may be simultaneously servicing other requests.

An arrival pattern for events may be characterized as either periodic or stochastic. For example, a periodic event may arrive every 10 milliseconds. Periodic event arrival is most often seen in real-time systems. Stochastic arrival means that events arrive according to some probabilistic distribution. Events can also arrive sporadically, that is, according to a pattern not capturable by either periodic or stochastic characterizations.

Multiple users or other loading factors can be modeled by varying the arrival pattern for events. In other words, from the point of view of system performance, it does not matter whether one user submits 20 requests in a period of time or whether two users each submit 10. What matters is the arrival pattern at the server and dependencies within the requests.

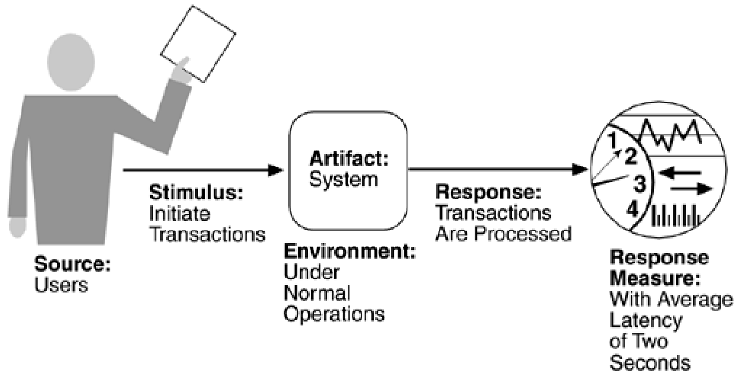
The response of the system to a stimulus can be characterized by latency (the time between the arrival of the stimulus and the system's response to it), deadlines in processing (in the engine controller, for example, the fuel should ignite when the cylinder is in a particular position, thus introducing a processing deadline), the throughput of the system (e.g., the number of transactions the system can process in a second), the jitter of the response (the variation in latency), the number of events not processed because the system was too busy to respond, and the data that was lost because the system was too busy.

Notice that this formulation does not consider whether the system is networked or standalone. Nor does it (yet) consider the configuration of the system or the consumption of resources. These issues are dependent on architectural solutions, which we will discuss in Chapter 5.

Performance General Scenarios

From these considerations we can see the portions of the performance general scenario, an example of which is shown in Figure 4.5: "Users initiate 1,000 transactions per minute stochastically under normal operations, and these transactions are processed with an average latency of two seconds."

Figure 4.5. Sample performance scenario



- Source of stimulus. The stimuli arrive either from external (possibly multiple) or internal sources. In our example, the source of the stimulus is a collection of users.
- Stimulus. The stimuli are the event arrivals. The arrival pattern can be characterized as periodic, stochastic, or sporadic. In our example, the stimulus is the stochastic initiation of 1,000 transactions per minute.
- Artifact. The artifact is always the system's services, as it is in our example.
- Environment. The system can be in various operational modes, such as normal, emergency, or overload. In our example, the system is in normal mode.
- Response. The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode). In our example, the transactions are processed.
- Response measure. The response measures are the time it takes to process the arriving events (latency or a deadline by which the event must be processed), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate, data loss). In our example, the transactions should be processed with an average latency of two seconds.

Table 4.3 gives elements of the general scenarios that characterize performance.

Table 4.3. Performance General Scenario Generation

Portion of Scenario	Possible Values
Source	One of a number of independent sources, possibly from within system
Stimulus	Periodic events arrive; sporadic events arrive; stochastic events arrive
Artifact	System
Environment	Normal mode; overload mode

Table 4.3. Performance General Scenario Generation

Portion of Scenario	Possible Values
Response	Processes stimuli; changes level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate, data loss

For most of the history of software engineering, performance has been the driving factor in system architecture. As such, it has frequently compromised the achievement of all other qualities. As the price/performance ratio of hardware plummets and the cost of developing software rises, other qualities have emerged as important competitors to performance.

SECURITY

Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users. An attempt to breach security is called an attack[1] and can take a number of forms. It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

[1] Some security experts use "threat" interchangeably with "attack."

Attacks, often occasions for wide media coverage, may range from theft of money by electronic transfer to modification of sensitive data, from theft of credit card numbers to destruction of files on computer systems, or to denial-of-service attacks carried out by worms or viruses. Still, the elements of a security general scenario are the same as the elements of our other general scenarios—a stimulus and its source, an environment, the target under attack, the desired response of the system, and the measure of this response.

Security can be characterized as a system providing nonrepudiation, confidentiality, integrity, assurance, availability, and auditing. For each term, we provide a definition and an example.

1. Nonrepudiation is the property that a transaction (access to or modification of data or services) cannot be denied by any of the parties to it. This means you cannot deny that you ordered that item over the Internet if, in fact, you did.
2. Confidentiality is the property that data or services are protected from unauthorized access. This means that a hacker cannot access your income tax returns on a government computer.
3. Integrity is the property that data or services are being delivered as intended. This means that your grade has not been changed since your instructor assigned it.
4. Assurance is the property that the parties to a transaction are who they purport to be. This means that, when a customer sends a credit card number to an Internet merchant, the merchant is who the customer thinks they are.
5. Availability is the property that the system will be available for legitimate use. This means that a denial-of-service attack won't prevent your ordering this book.
6. Auditing is the property that the system tracks activities within it at levels sufficient to reconstruct them. This means that, if you transfer money out of one account to another account, in Switzerland, the system will maintain a record of that transfer.

Each of these security categories gives rise to a collection of general scenarios.

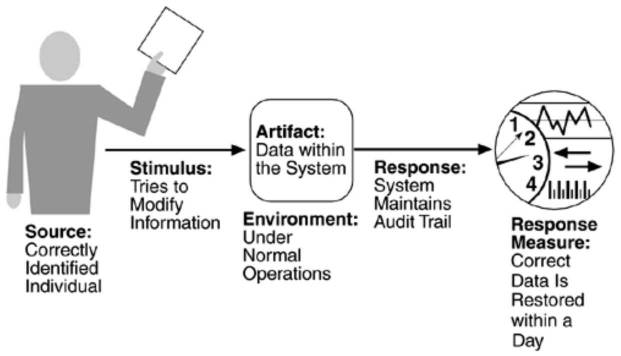
Security General Scenarios

The portions of a security general scenario are given below. Figure 4.6 presents an example. A correctly identified individual tries to modify system data from an external site; system maintains an audit trail and the correct data is restored within one day.

- Source of stimulus. The source of the attack may be either a human or another system. It may have been previously identified (either correctly or incorrectly) or may be currently unknown. If the source of the attack is highly motivated (say politically motivated), then defensive measures such as "We know who you are and will prosecute you" are not likely to be effective; in such cases the motivation of the user may be important. If the source has access to vast resources (such as a government), then defensive measures are very difficult. The attack itself is unauthorized access, modification, or denial of service.

  The difficulty with security is allowing access to legitimate users and determining legitimacy. If the only goal were to prevent access to a system, disallowing all access would be an effective defensive measure.

Figure 4.6. Sample security scenario


Source: Correctly Identified Individual
Stimulus: Tries to Modify Information
Artifact: Data within the System
Environment: Under Normal Operations
Response: System Maintains Audit Trail
Response Measure: Correct Data Is Restored within a Day
1
2
3
4

- Stimulus. The stimulus is an attack or an attempt to break security. We characterize this as an unauthorized person or system trying to display information, change and/or delete information, access services of the system, or reduce availability of system services. In Figure 4.6, the stimulus is an attempt to modify data.
- Artifact. The target of the attack can be either the services of the system or the data within it. In our example, the target is data within the system.
- Environment. The attack can come when the system is either online or offline, either connected to or disconnected from a network, either behind a firewall or open to the network.
- Response. Using services without authorization or preventing legitimate users from using services is a different goal from seeing sensitive data or modifying it. Thus, the system must authorize legitimate users and grant them access to data and services, at the same time rejecting unauthorized users, denying them access, and reporting unauthorized access. Not only does the system need to provide access to legitimate users, but it needs to support the granting or withdrawing of access. One technique to prevent attacks is to cause fear of punishment by maintaining an audit trail of modifications or attempted accesses. An audit trail is also useful in correcting from a successful attack. In Figure 4.6, an audit trail is maintained.
- Response measure. Measures of a system's response include the difficulty of mounting various attacks and the difficulty of recovering from and surviving attacks. In our example, the audit trail allows the accounts from which money was embezzled to be restored to their original state. Of course, the embezzler still has the money, and he must be tracked down and the money regained, but this is outside of the realm of the computer system.

Table 4.4 shows the security general scenario generation table.

Table 4.4. Security General Scenario Generation

Portion of Scenario	**Possible Values**
Source	Individual or system that is correctly identified, identified incorrectly, of unknown identity who is internal/external, authorized/not authorized with access to limited resources, vast resources
Stimulus	Tries to display data, change/delete data, access system services, reduce availability to system services
Artifact	System services; data within system
Environment	Either online or offline, connected or disconnected, firewalled or open
Response	Authenticates user; hides identity of the user; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a user or another system, and restricts availability of services
Response Measure	Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied

TESTABILITY

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. At least 40% of the cost of developing well-engineered systems is taken up by testing. If the software architect can reduce this cost, the payoff is large.

In particular, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution. Of course, calculating this probability is not easy and, when we get to response measures, other measures will be used.

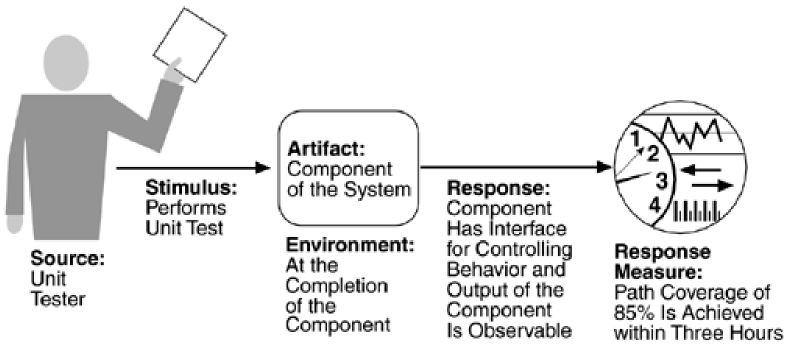
For a system to be properly testable, it must be possible to control each component's internal state and inputs and then to observe its outputs. Frequently this is done through use of a test harness, specialized software designed to exercise the software under test. This may be as simple as a playback capability for data recorded across various interfaces or as complicated as a testing chamber for an engine.

Testing is done by various developers, testers, verifiers, or users and is the last step of various parts of the software life cycle. Portions of the code, the design, or the complete system may be tested. The response measures for testability deal with how effective the tests are in discovering faults and how long it takes to perform the tests to some desired level of coverage.

Testability General Scenarios

Figure 4.7 is an example of a testability scenario concerning the performance of a unit test: A unit tester performs a unit test on a completed system component that provides an interface for controlling its behavior and observing its output; 85% path coverage is achieved within three hours.

Figure 4.7. Sample testability scenario


Source:
Unit Tester
Stimulus:
Performs Unit Test
Artifact:
Component of the System
Environment:
At the Completion of the Component
Response:
Component Has Interface for Controlling Behavior and Output of the Component Is Observable
Response Measure:
Path Coverage of 85% Is Achieved within Three Hours

- Source of stimulus. The testing is performed by unit testers, integration testers, system testers, or the client. A test of the design may be performed by other developers or by an external group. In our example, the testing is performed by a tester.
- Stimulus. The stimulus for the testing is that a milestone in the development process is met. This might be the completion of an analysis or design increment, the completion of a coding increment such as a class, the completed integration of a subsystem, or the completion of the whole system. In our example, the testing is triggered by the completion of a unit of code.
- Artifact. A design, a piece of code, or the whole system is the artifact being tested. In our example, a unit of code is to be tested.
- Environment. The test can happen at design time, at development time, at compile time, or at deployment time. In Figure 4.7, the test occurs during development.
- Response. Since testability is related to observability and controllability, the desired response is that the system can be controlled to perform the desired tests and that the response to each test can be observed. In our example, the unit can be controlled and its responses captured.
- Response measure. Response measures are the percentage of statements that have been executed in some test, the length of the longest test chain (a measure of the difficulty of performing the tests), and estimates of the probability of finding additional faults. In Figure 4.7, the measurement is percentage coverage of executable statements.

Table 4.5 gives the testability general scenario generation table.

Table 4.5. Testability General Scenario Generation

Portion of Scenario	Possible Values
Source	Unit developer Increment integrator System verifier Client acceptance tester System user

Table 4.5. Testability General Scenario Generation

Portion of Scenario	**Possible Values**
Stimulus	Analysis, architecture, design, class, subsystem integration completed; system delivered
Artifact	Piece of design, piece of code, complete application
Environment	At design time, at development time, at compile time, at deployment time
Response	Provides access to state values; provides computed values; prepares test environment
Response Measure	Percent executable statements executed Probability of failure if fault exists Time to perform tests Length of longest dependency chain in a test Length of time to prepare test environment

USABILITY

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides. It can be broken down into the following areas:

- Learning system features. If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier?
- Using a system efficiently. What can the system do to make the user more efficient in its operation?
- Minimizing the impact of errors. What can the system do so that a user error has minimal impact?
- Adapting the system to user needs. How can the user (or the system itself) adapt to make the user's task easier?
- Increasing confidence and satisfaction. What does the system do to give the user confidence that the correct action is being taken?

In the last five years, our understanding of the relation between usability and software architecture has deepened (see the sidebar Usability Mea Culpa). The normal development process detects usability problems through building prototypes and user testing. The later a problem is discovered and the deeper into the architecture its repair must be made, the more the repair is threatened by time and budget pressures. In our scenarios we focus on aspects of usability that have a major impact on the architecture. Consequently, these scenarios must be correct prior to the architectural design so that they will not be discovered during user testing or prototyping.

Usability General Scenarios

Figure 4.8 gives an example of a usability scenario: A user, wanting to minimize the impact of an error, wishes to cancel a system operation at runtime; cancellation takes place in less than one second. The portions of the usability general scenarios are:

- Source of stimulus. The end user is always the source of the stimulus.
- Stimulus. The stimulus is that the end user wishes to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or feel comfortable with the system. In our example, the user wishes to cancel an operation, which is an example of minimizing the impact of errors.
- Artifact. The artifact is always the system.

Usability Mea Culpa (or "That's Not Architectural")

About five years ago a number of respected software engineering researchers publicly made the following bold statement:

Making a system's user interface clear and easy to use is primarily a matter of getting the details of a user's interaction correct … but these details are not architectural.

Sad to say, these researchers were Bass, Clements, and Kazman, and the book was the first edition of Software Architecture in Practice. In the intervening five years we have learned quite a lot about many quality attributes, and none more so than usability.

While we have always claimed that system quality stems primarily from architectural quality, in the first edition of this book we were, at times, on shaky ground in trying to substantiate this claim. Still, the intervening years have done nothing to lessen the basic truth of the strong relationship between architectural quality and system quality. In fact, all of the evidence points squarely in its favor, and usability has proven to be no exception. Many usability issues are architectural. In fact, the usability features that are the most difficult to achieve (and, in particular, the most difficult to add on after the system has been built) turn out to be precisely those that are architectural.

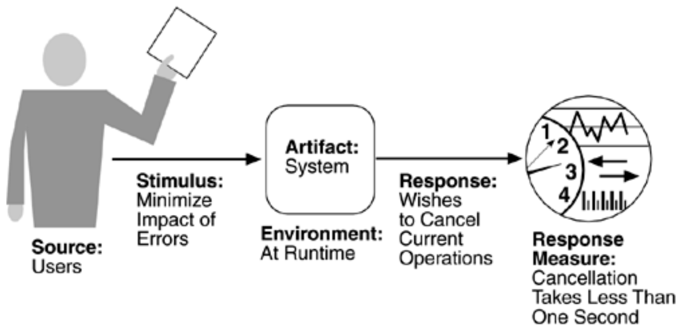
If you want to support the ability of a user to cancel an operation in progress, returning to the precise system state in place before the operation was started, you need to plan for this capability in the architecture. Likewise, if you want to support the ability of a user to undo a previous action and if you want to give the user feedback as to an operation's progress. There are many other examples.

The point here is that it is easy to assume that a quality attribute, or significant portions of a quality attribute, are not architectural. Not everything is architectural it's true, but frequently our assumptions of what is and what is not are based on a superficial analysis of the problem. Probe more deeply, and significant architectural considerations pop up everywhere. And woe to the architect (or architecture writer!) who ignores them.

— RK

- Environment. The user actions with which usability is concerned always occur at runtime or at system configuration time. Any action that occurs before then is performed by developers and, although a user may also be the developer, we distinguish between these roles even if performed by the same person. In Figure 4.8, the cancellation occurs at runtime.
- Response. The system should either provide the user with the features needed or anticipate the user's needs. In our example, the cancellation occurs as the user wishes and the system is restored to its prior state.
- Response measure. The response is measured by task time, number of errors, number of problems solved, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time/data lost when an error occurs. In Figure 4.8, the cancellation should occur in less than one second.

Figure 4.8. Sample usability scenario


Source:
Users
Stimulus:
Minimize Impact of Errors
Artifact:
System
Environment:
At Runtime
Response:
Wishes to Cancel Current Operations
Response Measure:
Cancellation Takes Less Than One Second

The usability general scenario generation table is given in Table 4.6.

Table 4.6. Usability General Scenario Generation

Portion of Scenario	**Possible Values**
Source	End user
Stimulus	Wants to learn system features; use system efficiently; minimize impact of errors; adapt system; feel comfortable
Artifact	System
Environment	At runtime or configure time
Response	System provides one or more of the following responses: to support "learn system features" help system is sensitive to context; interface is familiar to user; interface is usable in an unfamiliar context to support "use system efficiently": aggregation of data and/or commands; re-use of already entered data and/or commands; support for efficient navigation within a screen; distinct views with consistent operations; comprehensive searching; multiple simultaneous activities to "minimize impact of errors": undo, cancel, recover from system failure, recognize and correct user error, retrieve forgotten password, verify system resources to "adapt system": customizability; internationalization to "feel comfortable": display system state; work at the user's pace
Response Measure	Task time, number of errors, number of problems solved, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, amount of time/data lost

COMMUNICATING CONCEPTS USING GENERAL SCENARIOS

One of the uses of general scenarios is to enable stakeholders to communicate. We have already pointed out that each attribute community has its own vocabulary to describe its basic concepts and that different terms can represent the same occurrence. This may lead to miscommunication. During a discussion of performance, for example, a stakeholder representing users may not realize that the latency of the response to events has anything to do with users. Facilitating this kind of understanding aids discussions of architectural decisions, particularly about tradeoffs.

Table 4.7. Quality Attribute Stimuli

Quality Attribute	**Stimulus**
Availability	Unexpected event, nonoccurrence of expected event
Modifiability	Request to add/delete/change/vary functionality, platform, quality attribute, or capacity

Table 4.7. Quality Attribute Stimuli

Quality Attribute	Stimulus
Performance	Periodic, stochastic, or sporadic
Security	Tries to display, modify, change/delete information, access, or reduce availability to system services
Testability	Completion of phase of system development
Usability	Wants to learn system features, use a system efficiently, minimize the impact of errors, adapt the system, feel comfortable

Table 4.7 gives the stimuli possible for each of the attributes and shows a number of different concepts. Some stimuli occur during runtime and others occur before. The problem for the architect is to understand which of these stimuli represent the same occurrence, which are aggregates of other stimuli, and which are independent. Once the relations are clear, the architect can communicate them to the various stakeholders using language that each comprehends. We cannot give the relations among stimuli in a general way because they depend partially on environment. A performance event may be atomic or may be an aggregate of other lower-level occurrences; a failure may be a single performance event or an aggregate. For example, it may occur with an exchange of severalmessages between a client and a server (culminating in an unexpected message), each of which is an atomic event from a performance perspective.

4.5. Other System Quality Attributes

We have discussed the quality attributes in a general fashion. A number of other attributes can be found in the attribute taxonomies in the research literature and in standard software engineering textbooks, and we have captured many of these in our scenarios. For example, scalability is often an important attribute, but in our discussion here scalability is captured by modifying system capacity—the number of users supported, for example. Portability is captured as a platform modification.

If some quality attribute—say interoperability—is important to your organization, it is reasonable to create your own general scenario for it. This simply involves filling out the six parts of the scenario generation framework: source, stimulus, environment, artifact, response, and response measure. For interoperability, a stimulus might be a request to interoperate with another system, a response might be a new interface or set of interfaces for the interoperation, and a response measure might be the difficulty in terms of time, the number of interfaces to be modified, and so forth.

4.6. Business Qualities

In addition to the qualities that apply directly to a system, a number of business quality goals frequently shape a system's architecture. These goals center on cost, schedule, market, and marketing considerations. Each suffers from the same ambiguity that system qualities have, and they need to be made specific with scenarios in order to make them suitable for influencing the design process and to be made testable. Here, we present them as generalities, however, and leave the generation of scenarios as one of our discussion questions.

- Time to market. If there is competitive pressure or a short window of opportunity for a system or product, development time becomes important. This in turn leads to pressure to buy or otherwise re-use existing elements. Time to market is often reduced by using prebuilt elements such as commercial off-the-shelf (COTS) products or elements re-used from previous projects. The ability to insert or deploy a subset of the system depends on the decomposition of the system into elements.

- Cost and benefit. The development effort will naturally have a budget that must not be exceeded. Different architectures will yield different development costs. For instance, an architecture that relies on technology (or expertise with a technology) not resident in the developing organization will be more expensive to realize than one that takes advantage of assets already inhouse. An architecture that is highly flexible will typically be more costly to build than one that is rigid (although it will be less costly to maintain and modify).
- Projected lifetime of the system. If the system is intended to have a long lifetime, modifiability, scalability, and portability become important. But building in the additional infrastructure (such as a layer to support portability) will usually compromise time to market. On the other hand, a modifiable, extensible product is more likely to survive longer in the marketplace, extending its lifetime.
- Targeted market. For general-purpose (mass-market) software, the platforms on which a system runs as well as its feature set will determine the size of the potential market. Thus, portability and functionality are key to market share. Other qualities, such as performance, reliability, and usability also play a role. To attack a large market with a collection of related products, a product line approach should be considered in which a core of the system is common (frequently including provisions for portability) and around which layers of software of increasing specificity are constructed. Such an approach will be treated in Chapter 14, which discusses software product lines.
- Rollout schedule. If a product is to be introduced as base functionality with many features released later, the flexibility and customizability of the architecture are important. Particularly, the system must be constructed with ease of expansion and contraction in mind.
- Integration with legacy systems. If the new system has to integrate with existing systems, care must be taken to define appropriate integration mechanisms. This property is clearly of marketing importance but has substantial architectural implications. For example, the ability to integrate a legacy system with an HTTP server to make it accessible from the Web has been a marketing goal in many corporations over the past decade. The architectural constraints implied by this integration must be analyzed.

4.7. Architecture Qualities

In addition to qualities of the system and qualities related to the business environment in which the system is being developed, there are also qualities directly related to the architecture itself that are important to achieve. We discuss three, again leaving the generation of specific scenarios to our discussion questions.

Conceptual integrity is the underlying theme or vision that unifies the design of the system at all levels. The architecture should do similar things in similar ways. Fred Brooks writes emphatically that a system's conceptual integrity is of overriding importance, and that systems without it fail:

I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas. [Brooks 75]

Brooks was writing primarily about the way systems appear to their users, but the point is equally valid for the architectural layout. What Brooks's idea of conceptual integrity does for the user, architectural integrity does for the other stakeholders, particularly developers and maintainers.

In Part Three, you will see a recommendation for architecture evaluation that requires the project being reviewed to make the architect available. If no one is identified with that role, it is a sign that conceptual integrity may be lacking.

Correctness and completeness are essential for the architecture to allow for all of the system's requirements and runtime resource constraints to be met. A formal evaluation, as prescribed in Part Three, is once again the architect's best hope for a correct and complete architecture.

Buildability allows the system to be completed by the available team in a timely manner and to be open to certain changes as development progresses. It refers to the ease of constructing a desired system and is achieved architecturally by paying careful attention to the decomposition into modules, judiciously assigning of those modules to development teams, and limiting the dependencies between the modules (and hence the teams). The goal is to maximize the parallelism that can occur in development.

Because buildability is usually measured in terms of cost and time, there is a relationship between it and various cost models. However, buildability is more complex than what is usually covered in cost models. A system is created from certain materials, and these materials are created using a variety of tools. For example, a user interface may be constructed from items in a user interface toolbox (called widgets or controls), and these widgets may be manipulated by a user interface builder. The widgets are the materials and the builder is the tool, so one element of buildability is the match between the materials that are to be used in the system and the tools that are available to manipulate them. Another aspect of buildability is knowledge about the problem to be solved. The rationale behind this aspect is to speed time to market and not force potential suppliers to invest in the understanding and engineering of a new concept. A design that casts a solution in terms of well-understood concepts is thus more buildable than one that introduces new concepts.

4.8. Summary

The qualities presented in this chapter represent those most often the goals of software architects. Since their definitions overlap, we chose to characterize them with general scenarios. We saw that qualities can be divided into those that apply to the system, those that apply to the business environment, and those that apply to the architecture itself.

In the next chapter, we will explore concrete architectural approaches for following the path from qualities to architecture.

Chapter 5. Achieving Qualities

with Felix Bachmann, Mark Klein, and Bill Wood
Note: Felix Bachmann, Mark Klein, and Bill Wood are senior members of the technical staff at the Software Engineering Institute.

Every good quality is noxious if unmixed.
—Ralph Waldo Emerson

Chapter 4 characterized a number of system quality attributes. That characterization was in terms of a collection of scenarios. Understanding what is meant by a quality attribute enables you to elicit the quality requirements but provides no help in understanding how to achieve them. In this chapter, we begin to provide that help. For each of the six system quality attributes that we elaborated in Chapter 4, we provide architectural guidance for their achievement. The tactics enumerated here do not cover all possible quality attributes, but we will see tactics for integrability in Chapter 8.

We are interested in how the architect achieves particular qualities. The quality requirements specify the responses of the software to realize business goals. Our interest is in the tactics used by the architect to create a design using design patterns, architectural patterns, or architectural strategies. For example, a business goal might be to create a product line. A means of achieving that goal is to allow variability in particular classes of functions.

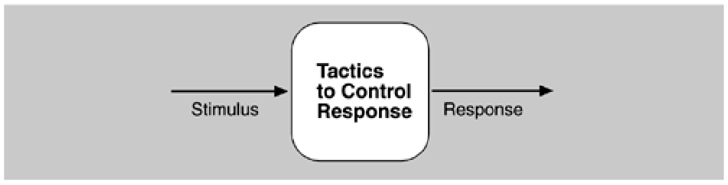
Prior to deciding on a set of patterns to achieve the desired variation, the architect should consider what combination of tactics for modifiability should be applied, as the tactics chosen will guide the architectural decisions. An architectural pattern or strategy implements a collection of tactics. The connection between quality attribute requirements (discussed in Chapter 4) and architectural decisions is the subject of this chapter.

5.1. Introducing Tactics

What is it that imparts portability to one design, high performance to another, and integrability to a third? The achievement of these qualities relies on fundamental design decisions. We will examine these design decisions, which we call tactics. A tactic is a design decision that influences the control of a quality attribute response. We call a collection of tactics an architectural strategy, which we will treat in Chapter 12. An architectural pattern packages tactics in a fashion that we will describe in Section 5.8.

A system design consists of a collection of decisions. Some of these decisions help control the quality attribute responses; others ensure achievement of system functionality. In this section, we discuss the quality attribute decisions known as tactics. We represent this relationship in Figure 5.1. The tactics are those that architects have been using for years, and we isolate and describe them. We are not inventing tactics here, just capturing what architects do in practice.

Figure 5.1. Tactics are intended to control responses to stimuli.



Each tactic is a design option for the architect. For example, one of the tactics introduces redundancy to increase the availability of a system. This is one option the architect has to increase availability, but not the only one. Usually achieving high availability through redundancy implies a concomitant need for synchronization (to ensure that the redundant copy can be used if the original fails). We see two immediate ramifications of this example.

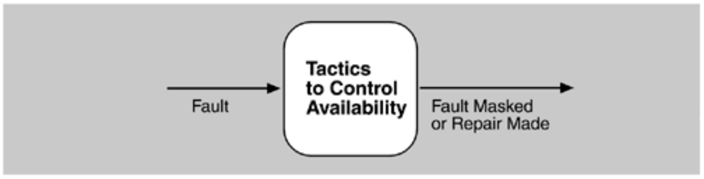
1. Tactics can refine other tactics. We identified redundancy as a tactic. As such, it can be refined into redundancy of data (in a database system) or redundancy of computation (in an embedded control system). Both types are also tactics. There are further refinements that a designer can employ to make each type of redundancy more concrete. For each quality attribute that we discuss, we organize the tactics as a hierarchy.
2. Patterns package tactics. A pattern that supports availability will likely use both a redundancy tactic and a synchronization tactic. It will also likely use more concrete versions of these tactics. At the end of this section, we present an example of a pattern described in terms of its tactics.

We organize the tactics for each system quality attribute as a hierarchy, but it is important to understand that each hierarchy is intended only to demonstrate some of the tactics, and that any list of tactics is necessarily incomplete. For each of the six attributes that we elaborated in Chapter 4 (availability, modifiability, performance, security, testability, and usability), we discuss tactical approaches for achieving it. For each, we present an organization of the tactics and a brief discussion. The organization is intended to provide a path for the architect to search for appropriate tactics.

5.2. Availability Tactics

Recall the vocabulary for availability from Chapter 4. A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's users. A fault (or combination of faults) has the potential to cause a failure. Recall also that recovery or repair is an important aspect of availability. The tactics we discuss in this section will keep faults from becoming failures or at least bound the effects of the fault and make repair possible. We illustrate this in Figure 5.2.

Figure 5.2. Goal of availability tactics



Many of the tactics we discuss are available within standard execution environments such as operating systems, application servers, and database management systems. It is still important to understand the tactics used so that the effects of using a particular one can be considered during design and evaluation. All approaches to maintaining availability involve some type of redundancy, some type of health monitoring to detect a failure, and some type of recovery when a failure is detected. In some cases, the monitoring or recovery is automatic and in others it is manual.

We first consider fault detection. We then consider fault recovery and finally, briefly, fault prevention.

FAULT DETECTION

Three widely used tactics for recognizing faults are ping/echo, heartbeat, and exceptions.

- Ping/echo. One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny. This can be used within a group of components mutually responsible for one task. It can also used be used by clients to ensure that a server object and the communication path to the server are operating within the expected performance bounds. "Ping/echo" fault detectors can be organized in a hierarchy, in which a lowest-level detector pings the software processes with which it shares a processor, and the higher-level fault detectors ping lower-level ones. This uses less communications bandwidth than a remote fault detector that pings all processes.

- Heartbeat (dead man timer). In this case one component emits a heartbeat message periodically and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified. The heartbeat can also carry data. For example, an automated teller machine can periodically send the log of the last transaction to a server. This message not only acts as a heartbeat but also carries data to be processed.
- Exceptions. One method for recognizing faults is to encounter an exception, which is raised when one of the fault classes we discussed in Chapter 4 is recognized. The exception handler typically executes in the same process that introduced the exception.

The ping/echo and heartbeat tactics operate among distinct processes, and the exception tactic operates within a single process. The exception handler will usually perform a semantic transformation of the fault into a form that can be processed.

FAULT RECOVERY

Fault recovery consists of preparing for recovery and making the system repair. Some preparation and repair tactics follow.

- Voting. Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter. If the voter detects deviant behavior from a single processor, it fails it. The voting algorithm can be "majority rules" or "preferred component" or some other algorithm. This method is used to correct faulty operation of algorithms or failure of a processor and is often used in control systems. If all of the processors utilize the same algorithms, the redundancy detects only a processor fault and not an algorithm fault. Thus, if the consequence of a failure is extreme, such as potential loss of life, the redundant components can be diverse.

  One extreme of diversity is that the software for each redundant component is developed by different teams and executes on dissimilar platforms. Less extreme is to develop a single software component on dissimilar platforms. Diversity is expensive to develop and maintain and is used only in exceptional circumstances, such as the control of surfaces on aircraft. It is usually used for control systems in which the outputs to the voter are straightforward and easy to classify as equivalent or deviant, the computations are cyclic, and all redundant components receive equivalent inputs from sensors. Diversity has no downtime when a failure occurs since the voter continues to operate. Variations on this approach include the Simplex approach, which uses the results of a "preferred" component unless they deviate from those of a "trusted" component, to which it defers. Synchronization among the redundant components is automatic since they are all assumed to be computing on the same set of inputs in parallel.

- Active redundancy (hot restart). All redundant components respond to events in parallel. Consequently, they are all in the same state. The response from only one component is used (usually the first to respond), and the rest are discarded. When a fault occurs, the downtime of systems using this tactic is usually milliseconds since the backup is current and the only time to recover is the switching time. Active redundancy is often used in a client/server configuration, such as database management systems, where quick responses are necessary even when a fault occurs. In a highly available distributed system, the redundancy may be in the communication paths. For example, it may be desirable to use a LAN with a number of parallel paths and place each redundant component in a separate path. In this case, a single bridge or path failure will not make all of the system's components unavailable.

Synchronization is performed by ensuring that all messages to any redundant component are sent to all redundant components. If communication has a possibility of being lost (because of noisy or overloaded communication lines), a reliable transmission protocol can be used to recover. A reliable transmission protocol requires all recipients to acknowledge receipt together with some integrity indication such as a checksum. If the sender cannot verify that all recipients have received the message, it will resend the message to those components not acknowledging receipt. The resending of unreceived messages (possibly over different communication paths) continues until the sender marks the recipient as out of service.

- Passive redundancy (warm restart/dual redundancy/triple redundancy). One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make. When a fault occurs, the system must first ensure that the backup state is sufficiently fresh before resuming services. This approach is also used in control systems, often when the inputs come over communication channels or from sensors and have to be switched from the primary to the backup on failure. Chapter 6, describing an air traffic control example, shows a system using it. In the air traffic control system, the secondary decides when to take over from the primary, but in other systems this decision can be done in other components. This tactic depends on the standby components taking over reliably. Forcing switchovers periodically—for example, once a day or once a week—increases the availability of the system. Some database systems force a switch with storage of every new data item. The new data item is stored in a shadow page and the old page becomes a backup for recovery. In this case, the downtime can usually be limited to seconds.

  Synchronization is the responsibility of the primary component, which may use atomic broadcasts to the secondaries to guarantee synchronization.

- Spare. A standby spare computing platform is configured to replace many different failed components. It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs. Making a checkpoint of the system state to a persistent device periodically and logging all state changes to a persistent device allows for the spare to be set to the appropriate state. This is often used as the standby client workstation, where the user can move when a failure occurs. The downtime for this tactic is usually minutes.

  There are tactics for repair that rely on component reintroduction. When a redundant component fails, it may be reintroduced after it has been corrected. Such tactics are shadow operation, state resynchronization, and rollback.

- Shadow operation. A previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behavior of the working components before restoring it to service.
- State resynchronization. The passive and active redundancy tactics require the component being restored to have its state upgraded before its return to service. The updating approach will depend on the downtime that can be sustained, the size of the update, and the number of messages required for the update. A single message containing the state is preferable, if possible. Incremental state upgrades, with periods of service between increments, lead to complicated software.
- Checkpoint/rollback. A checkpoint is a recording of a consistent state created either periodically or in response to specific events. Sometimes a system fails in an unusual manner, with a detectably inconsistent state. In this case, the system should be restored using a previous checkpoint of a consistent state and a log of the transactions that occurred since the snapshot was taken.

FAULT PREVENTION

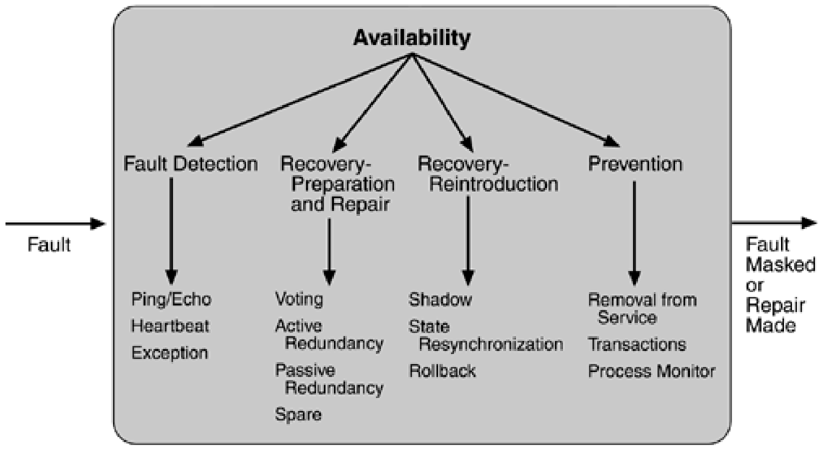
The following are some fault prevention tactics.

- Removal from service. This tactic removes a component of the system from operation to undergo some activities to prevent anticipated failures. One example is rebooting a component to prevent memory leaks from causing a failure. If this removal from service is automatic, an architectural strategy can be designed to support it. If it is manual, the system must be designed to support it.

- Transactions. A transaction is the bundling of several sequential steps such that the entire bundle can be undone at once. Transactions are used to prevent any data from being affected if one step in a process fails and also to prevent collisions among several simultaneous threads accessing the same data.
- Process monitor. Once a fault in a process has been detected, a monitoring process can delete the nonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.

Figure 5.3 summarizes the tactics just discussed.

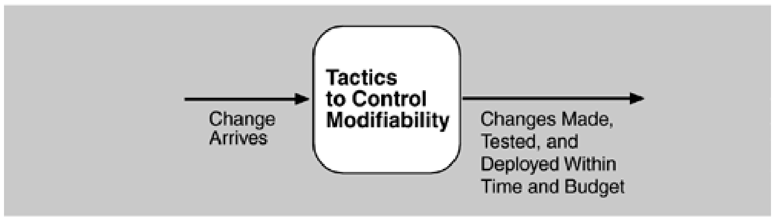
Figure 5.3. Summary of availability tactics



5.3. Modifiability Tactics

Recall from Chapter 4 that tactics to control modifiability have as their goal controlling the time and cost to implement, test, and deploy changes. Figure 5.4 shows this relationship.

Figure 5.4. Goal of modifiability tactics


Change Arrives
Tactics to Control Modifiability
Changes Made, Tested, and Deployed Within Time and Budget

We organize the tactics for modifiability in sets according to their goals. One set has as its goal reducing the number of modules that are directly affected by a change. We call this set "localize modifications." A second set has as its goal limiting modifications to the localized modules. We use this set of tactics to "prevent the ripple effect." Implicit in this distinction is that there are modules directly affected (those whose responsibilities are adjusted to accomplish the change) and modules indirectly affected by a change (those whose responsibilities remain unchanged but whose implementation must be changed to accommodate the directly affected modules). A third set of tactics has as its goal controlling deployment time and cost. We call this set "defer binding time."

LOCALIZE MODIFICATIONS

Although there is not necessarily a precise relationship between the number of modules affected by a set of changes and the cost of implementing those changes, restricting modifications to a small set of modules will generally reduce the cost. The goal of tactics in this set is to assign responsibilities to modules during design such that anticipated changes will be limited in scope. We identify five such tactics.

- Maintain semantic coherence. Semantic coherence refers to the relationships among responsibilities in a module. The goal is to ensure that all of these responsibilities work together without excessive reliance on other modules. Achievement of this goal comes from choosing responsibilities that have semantic coherence. Coupling and cohesion metrics are an attempt to measure semantic coherence, but they are missing the context of a change. Instead, semantic coherence should be measured against a set of anticipated changes. One subtactic is to abstract common services. Providing common services through specialized modules is usually viewed as supporting re-use. This is correct, but abstracting common services also supports modifiability. If common services have been abstracted, modifications to them will need to be made only once rather than in each module where the services are used. Furthermore, modification to the modules using those services will not impact other users. This tactic, then, supports not only localizing modifications but also the prevention of ripple effects. Examples of abstracting common services are the use of application frameworks and the use of other middleware software.
- Anticipate expected changes. Considering the set of envisioned changes provides a way to evaluate a particular assignment of responsibilities. The basic question is "For each change, does the proposed decomposition limit the set of modules that need to be modified to accomplish it?" An associated question is "Do fundamentally different changes affect the same modules?" How is this different from semantic coherence? Assigning responsibilities based on semantic coherence assumes that expected changes will be semantically coherent. The tactic of anticipating expected changes does not concern itself with the coherence of a module's responsibilities but rather with minimizing the effects of the changes. In reality this tactic is difficult to use by itself since it is not possible to anticipate all changes. For that reason, it is usually used in conjunction with semantic coherence.
- Generalize the module. Making a module more general allows it to compute a broader range of functions based on input. The input can be thought of as defining a language for the module, which can be as simple as making constants input parameters or as complicated as implementing the module as an interpreter and making the input parameters be a program in the interpreter's language. The more general a module, the more likely that requested changes can be made by adjusing the input language rather than by modifying the module.
- Limit possible options. Modifications, especially within a product line (see Chapter 14), may be far ranging and hence affect many modules. Restricting the possible options will reduce the effect of these modifications. For example, a variation point in a product line may be allowing for a change of processor. Restricting processor changes to members of the same family limits the possible options.

PREVENT RIPPLE EFFECTS

A ripple effect from a modification is the necessity of making changes to modules not directly affected by it. For instance, if module A is changed to accomplish a particular modification, then module B is changed only because of the change to module A. B has to be modified because it depends, in some sense, on A.

We begin our discussion of the ripple effect by discussing the various types of dependencies that one module can have on another. We identify eight types:

1. Syntax of

   - data. For B to compile (or execute) correctly, the type (or format) of the data that is produced by A and consumed by B must be consistent with the type (or format) of data assumed by B.

- service. For B to compile and execute correctly, the signature of services provided by A and invoked by B must be consistent with the assumptions of B.

2. Semantics of

   - data. For B to execute correctly, the semantics of the data produced by A and consumed by B must be consistent with the assumptions of B.

   - service. For B to execute correctly, the semantics of the services produced by A and used by B must be consistent with the assumptions of B.

3. Sequence of

   - data. For B to execute correctly, it must receive the data produced by A in a fixed sequence. For example, a data packet's header must precede its body in order of reception (as opposed to protocols that have the sequence number built into the data).

   - control. For B to execute correctly, A must have executed previously within certain timing constraints. For example, A must have executed no longer than 5ms before B executes.

4. Identity of an interface of A. A may have multiple interfaces. For B to compile and execute correctly, the identity (name or handle) of the interface must be consistent with the assumptions of B.
5. Location of A (runtime). For B to execute correctly, the runtime location of A must be consistent with the assumptions of B. For example, B may assume that A is located in a different process on the same processor.
6. Quality of service/data provided by A. For B to execute correctly, some property involving the quality of the data or service provided by A must be consistent with B's assumptions. For example, data provided by a particular sensor must have a certain accuracy in order for the algorithms of B to work correctly.
7. Existence of A. For B to execute correctly, A must exist. For example, if B is requesting a service from an object A, and A does not exist and cannot be dynamically created, then B will not execute correctly.
8. Resource behavior of A. For B to execute correctly, the resource behavior of A must be consistent with B's assumptions. This can be either resource usage of A (A uses the same memory as B) or resource ownership (B reserves a resource that A believes it owns).

With this understanding of dependency types, we can now discuss tactics available to the architect for preventing the ripple effect for certain types.

Notice that none of our tactics necessarily prevent the ripple of semantic changes. We begin with discussion of those that are relevant to the interfaces of a particular module—information hiding and maintaining existing interfaces—and follow with one that breaks a dependency chain—use of an intermediary.

- Hide information. Information hiding is the decomposition of the responsibilities for an entity (a system or some decomposition of a system) into smaller pieces and choosing which information to make private and which to make public. The public responsibilities are available through specified interfaces. The goal is to isolate changes within one module and prevent changes from propagating to others. This is the oldest technique for preventing changes from propagating. It is strongly related to "anticipate expected changes" because it uses those changes as the basis for decomposition.
- Maintain existing interfaces. If B depends on the name and signature of an interface of A, maintaining this interface and its syntax allows B to remain unchanged. Of course, this tactic will not necessarily work if B has a semantic dependency on A, since changes to the meaning of data and

services are difficult to mask. Also, it is difficult to mask dependencies on quality of data or quality of service, resource usage, or resource ownership. Interface stability can also be achieved by separating the interface from the implementation. This allows the creation of abstract interfaces that mask variations. Variations can be embodied within the existing responsibilities, or they can be embodied by replacing one implementation of a module with another.

Patterns that implement this tactic include

- adding interfaces. Most programming languages allow multiple interfaces. Newly visible services or data can be made available through new interfaces, allowing existing interfaces to remain unchanged and provide the same signature.

- adding adapter. Add an adapter to A that wraps A and provides the signature of the original A.

- providing a stub A. If the modification calls for the deletion of A, then providing a stub for A will allow B to remain unchanged if B depends only on A's signature.

- Restrict communication paths. Restrict the modules with which a given module shares data. That is, reduce the number of modules that consume data produced by the given module and the number of modules that produce data consumed by it. This will reduce the ripple effect since data production/consumption introduces dependencies that cause ripples. Chapter 8 (Flight Simulation) discusses a pattern that uses this tactic.
- Use an intermediary. If B has any type of dependency on A other than semantic, it is possible to insert an intermediary between B and A that manages activities associated with the dependency. All of these intermediaries go by different names, but we will discuss each in terms of the dependency types we have enumerated. As before, in the worst case, an intermediary cannot compensate for semantic changes. The intermediaries are

  - data (syntax). Repositories (both blackboard and passive) act as intermediaries between the producer and consumer of data. The repositories can convert the syntax produced by A into that assumed by B. Some publish/subscribe patterns (those that have data flowing through a central component) can also convert the syntax into that assumed by B. The MVC and PAC patterns convert data in one formalism (input or output device) into another (that used by the model in MVC or the abstraction in PAC).

  - service (syntax). The facade, bridge, mediator, strategy, proxy, and factory patterns all provide intermediaries that convert the syntax of a service from one form into another. Hence, they can all be used to prevent changes in A from propagating to B.

  - identity of an interface of A. A broker pattern can be used to mask changes in the identity of an interface. If B depends on the identity of an interface of A and that identity changes, by adding that identity to the broker and having the broker make the connection to the new identity of A, B can remain unchanged.

  - location of A (runtime). A name server enables the location of A to be changed without affecting B. A is responsible for registering its current location with the name server, and B retrieves that location from the name server.

  - resource behavior of A or resource controlled by A. A resource manager is an intermediary that is responsible for resource allocation. Certain resource managers (e.g., those based on Rate

Monotonic Analysis in real-time systems) can guarantee the satisfaction of all requests within certain constraints. A, of course, must give up control of the resource to the resource manager.

- existence of A. The factory pattern has the ability to create instances as needed, and thus the dependence of B on the existence of A is satisfied by actions of the factory.

DEFER BINDING TIME

The two tactic categories we have discussed thus far are designed to minimize the number of modules that require changing to implement modifications. Our modifiability scenarios include two elements that are not satisfied by reducing the number of modules to be changed—time to deploy and allowing nondevelopers to make changes. Deferring binding time supports both of those scenarios at the cost of requiring additional infrastructure to support the late binding.

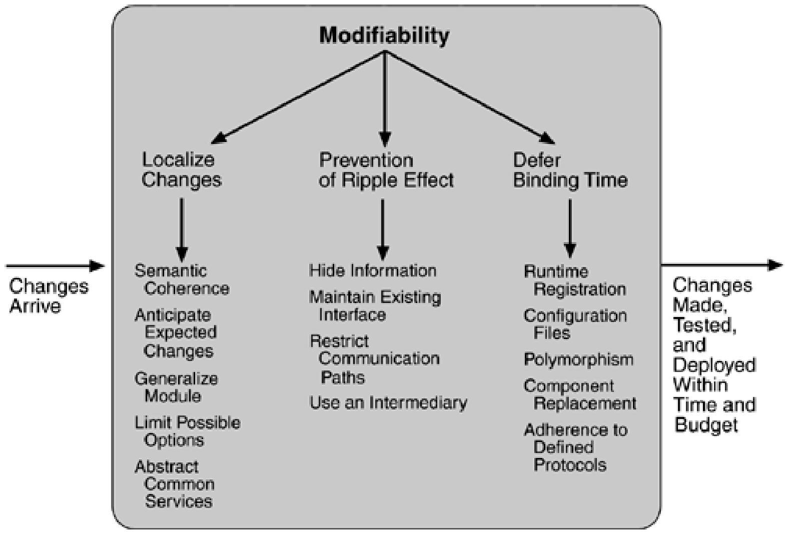
Decisions can be bound into the executing system at various times. We discuss those that affect deployment time. The deployment of a system is dictated by some process. When a modification is made by the developer, there is usually a testing and distribution process that determines the time lag between the making of the change and the availability of that change to the end user. Binding at runtime means that the system has been prepared for that binding and all of the testing and distribution steps have been completed. Deferring binding time also supports allowing the end user or system administrator to make settings or provide input that affects behavior.

Many tactics are intended to have impact at loadtime or runtime, such as the following.

- Runtime registration supports plug-and-play operation at the cost of additional overhead to manage the registration. Publish/subscribe registration, for example, can be implemented at either runtime or load time.
- Configuration files are intended to set parameters at startup.
- Polymorphism allows late binding of method calls.
- Component replacement allows load time binding.
- Adherence to defined protocols allows runtime binding of independent processes.

The tactics for modifiability are summarized in Figure 5.5.

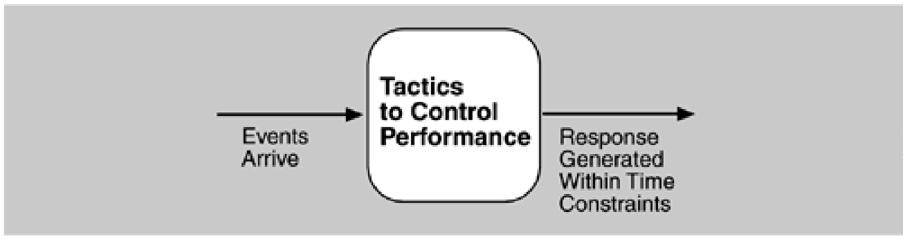
Figure 5.5. Summary of modifiability tactics



5.4. Performance Tactics

Recall from Chapter 4 that the goal of performance tactics is to generate a response to an event arriving at the system within some time constraint. The event can be single or a stream and is the trigger for a request to perform computation. It can be the arrival of a message, the expiration of a time interval, the detection of a significant change of state in the system's environment, and so forth. The system processes the events and generates a response. Performance tactics control the time within which a response is generated. This is shown in Figure 5.6. Latency is the time between the arrival of an event and the generation of a response to it.

Figure 5.6. Goal of performance tactics


Events Arrive
Tactics to Control Performance
Response Generated Within Time Constraints

After an event arrives, either the system is processing on that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: resource consumption and blocked time.

1. Resource consumption. Resources include CPU, data stores, network communication bandwidth, and memory, but it can also include entities defined by the particular system under design. For example, buffers must be managed and access to critical sections must be made sequential. Events can be of varying types (as just enumerated), and each type goes through a processing sequence. For example, a message is generated by one component, is placed on the network, and arrives at another component. It is then placed in a buffer; transformed in some fashion (marshalling is the term the Object Management Group uses for this transformation); processed according to some algorithm;

transformed for output; placed in an output buffer; and sent onward to another component, another system, or the user. Each of these phases contributes to the overall latency of the processing of that event.

2. Blocked time. A computation can be blocked from using a resource because of contention for it, because the resource is unavailable, or because the computation depends on the result of other computations that are not yet available.

   - Contention for resources. Figure 5.6 shows events arriving at the system. These events may be in a single stream or in multiple streams. Multiple streams vying for the same resource or different events in the same stream vying for the same resource contribute to latency. In general, the more contention for a resource, the more likelihood of latency being introduced. However, this depends on how the contention is arbitrated and how individual requests are treated by the arbitration mechanism.

   - Availability of resources. Even in the absence of contention, computation cannot proceed if a resource is unavailable. Unavailability may be caused by the resource being offline or by failure of the component or for some other reason. In any case, the architect must identify places where resource unavailability might cause a significant contribution to overall latency.

   - Dependency on other computation. A computation may have to wait because it must synchronize with the results of another computation or because it is waiting for the results of a computation that it initiated. For example, it may be reading information from two different sources, if these two sources are read sequentially, the latency will be higher than if they are read in parallel.

With this background, we turn to our three tactic categories: resource demand, resource management, and resource arbitration.

RESOURCE DEMAND

Event streams are the source of resource demand. Two characteristics of demand are the time between events in a resource stream (how often a request is made in a stream) and how much of a resource is consumed by each request.

One tactic for reducing latency is to reduce the resources required for processing an event stream. Ways to do this include the following.

- Increase computational efficiency. One step in the processing of an event or a message is applying some algorithm. Improving the algorithms used in critical areas will decrease latency. Sometimes one resource can be traded for another. For example, intermediate data may be kept in a repository or it may be regenerated depending on time and space resource availability. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk.
- Reduce computational overhead. If there is no request for a resource, processing needs are reduced. In Chapter 17, we will see an example of using Java classes rather than Remote Method Invocation (RMI) because the former reduces communication requirements. The use of intermediaries (so important for modifiability) increases the resources consumed in processing an event stream, and so removing them improves latency. This is a classic modifiability/performance tradeoff.

Another tactic for reducing latency is to reduce the number of events processed. This can be done in one of two fashions.

- Manage event rate. If it is possible to reduce the sampling frequency at which environmental variables are monitored, demand can be reduced. Sometimes this is possible if the system was overengineered. Other times an unnecessarily high sampling rate is used to establish harmonic periods between multiple streams. That is, some stream or streams of events are oversampled so that they can be synchronized.
- Control frequency of sampling. If there is no control over the arrival of externally generated events, queued requests can be sampled at a lower frequency, possibly resulting in the loss of requests.

Other tactics for reducing or managing demand involve controlling the use of resources.

- Bound execution times. Place a limit on how much execution time is used to respond to an event. Sometimes this makes sense and sometimes it does not. For iterative, data-dependent algorithms, limiting the number of iterations is a method for bounding execution times.
- Bound queue sizes. This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals.

RESOURCE MANAGEMENT

Even though the demand for resources may not be controllable, the management of these resources affects response times. Some resource management tactics are:

- Introduce concurrency. If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities. Once concurrency has been introduced, appropriately allocating the threads to resources (load balancing) is important in order to maximally exploit the concurrency.
- Maintain multiple copies of either data or computations. Clients in a client-server pattern are replicas of the computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a central server. Caching is a tactic in which data is replicated, either on different speed repositories or on separate repositories, to reduce contention. Since the data being cached is usually a copy of existing data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume.
- Increase available resources. Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency. Cost is usually a consideration in the choice of resources, but increasing the resources is definitely a tactic to reduce latency. This kind of cost/performance tradeoff is analyzed in Chapter 12.

RESOURCE ARBITRATION

Whenever there is contention for a resource, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled. The architect's goal is to understand the characteristics of each resource's use and choose the scheduling strategy that is compatible with it.

A scheduling policy conceptually has two parts: a priority assignment and dispatching. All scheduling policies assign priorities. In some cases the assignment is as simple as first-in/first-out. In other cases, it can be tied to the deadline of the request or its semantic importance. Competing criteria for scheduling include optimal resource usage, request importance, minimizing the number of resources used, minimizing latency, maximizing throughput, preventing starvation to ensure fairness, and so forth. The architect needs to be aware of these possibly conflicting criteria and the effect that the chosen tactic has on meeting them.

A high-priority event stream can be dispatched only if the resource to which it is being assigned is available. Sometimes this depends on pre-empting the current user of the resource. Possible preemption options are as follows: can occur anytime; can occur only at specific pre-emption points; and executing processes cannot be pre-empted. Some common scheduling policies are:

1. First-in/First-out. FIFO queues treat all requests for resources as equals and satisfy them in turn. One possibility with a FIFO queue is that one request will be stuck behind another one that takes a long time to generate a response. As long as all of the requests are truly equal, this is not a problem, but if some requests are of higher priority than others, it is problematic.
2. Fixed-priority scheduling. Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order. This strategy insures better service for higher-priority requests but admits the possibility of a low-priority, but important, request taking an arbitrarily long time to be serviced because it is stuck behind a series of higher-priority requests. Three common prioritization strategies are

   - semantic importance. Each stream is assigned a priority statically according to some domain characteristic of the task that generates it. This type of scheduling is used in mainframe systems where the domain characteristic is the time of task initiation.

   - deadline monotonic. Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines. This scheduling policy is used when streams of different priorities with real-time deadlines are to be scheduled.

   - rate monotonic. Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods. This scheduling policy is a special case of deadline monotonic but is better known and more likely to be supported by the operating system.

3. Dynamic priority scheduling:

   - round robin. Round robin is a scheduling strategy that orders the requests and then, at every assignment possibility, assigns the resource to the next request in that order. A special form of round robin is a cyclic executive where assignment possibilities are at fixed time intervals.

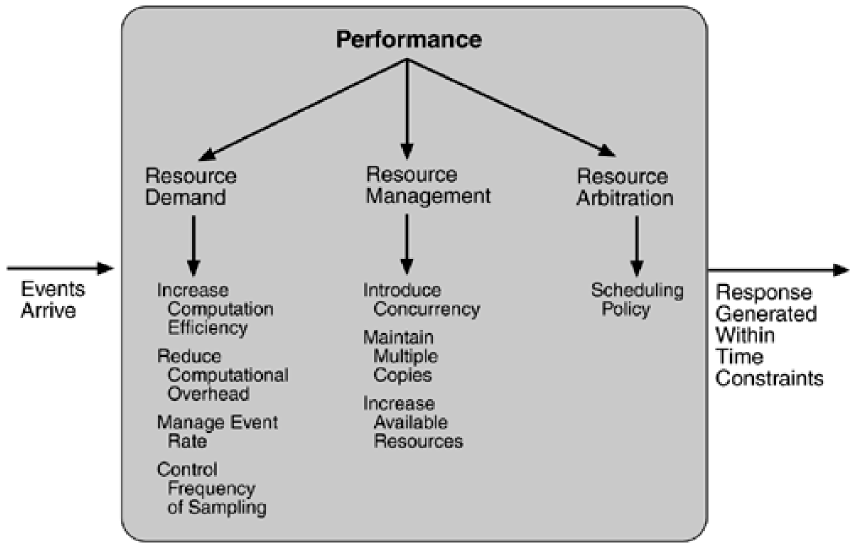
   - earliest deadline first. Earliest deadline first assigns priorities based on the pending requests with the earliest deadline.

4. Static scheduling. A cyclic executive schedule is a scheduling strategy where the pre-emption points and the sequence of assignment to the resource are determined offline.

For Further Reading at the end of this chapter lists books on scheduling theory.

The tactics for performance are summarized in Figure 5.7.

Figure 5.7. Summary of performance tactics

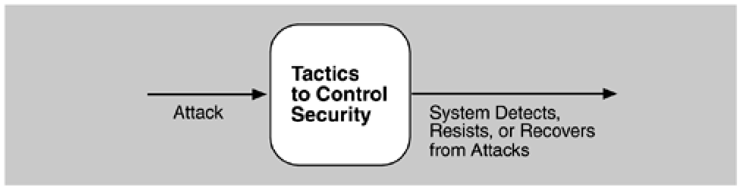


Performance
Resource Demand
Resource Management
Resource Arbitration
Events Arrive
Increase Computation Efficiency
Reduce Computational Overhead
Manage Event Rate
Control Frequency of Sampling
Introduce Concurrency
Maintain Multiple Copies
Increase Available Resources
Scheduling Policy
Response Generated Within Time Constraints

5.5. Security Tactics

Tactics for achieving security can be divided into those concerned with resisting attacks, those concerned with detecting attacks, and those concerned with recovering from attacks. All three categories are important. Using a familiar analogy, putting a lock on your door is a form of resisting an attack, having a motion sensor inside of your house is a form of detecting an attack, and having insurance is a form of recovering from an attack. Figure 5.8 shows the goals of the security tactics.

Figure 5.8. Goal of security tactics



Attack
Tactics to Control Security
System Detects, Resists, or Recovers from Attacks

RESISTING ATTACKS

In Chapter 4, we identified nonrepudiation, confidentiality, integrity, and assurance as goals in our security characterization. The following tactics can be used in combination to achieve these goals.

- Authenticate users. Authentication is ensuring that a user or remote computer is actually who it purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications provide authentication.
- Authorize users. Authorization is ensuring that an authenticated user has the rights to access and modify either data or services. This is usually managed by providing some access control patterns within a system. Access control can be by user or by user class. Classes of users can be defined by user groups, by user roles, or by lists of individuals.
- Maintain data confidentiality. Data should be protected from unauthorized access. Confidentiality is usually achieved by applying some form of encryption to data and to communication links. Encryption provides extra protection to persistently maintained data beyond that available from

authorization. Communication links, on the other hand, typically do not have authorization controls. Encryption is the only protection for passing data over publicly accessible communication links. The link can be implemented by a virtual private network (VPN) or by a Secure Sockets Layer (SSL) for a Web-based link. Encryption can be symmetric (both parties use the same key) or asymmetric (public and private keys).

- Maintain integrity. Data should be delivered as intended. It can have redundant information encoded in it, such as checksums or hash results, which can be encrypted either along with or independently from the original data.
- Limit exposure. Attacks typically depend on exploiting a single weakness to attack all data and services on a host. The architect can design the allocation of services to hosts so that limited services are available on each host.
- Limit access. Firewalls restrict access based on message source or destination port. Messages from unknown sources may be a form of an attack. It is not always possible to limit access to known sources. A public Web site, for example, can expect to get requests from unknown sources. One configuration used in this case is the so-called demilitarized zone (DMZ). A DMZ is used when access must be provided to Internet services but not to a private network. It sits between the Internet and a firewall in front of the internal network. The DMZ contains devices expected to receive messages from arbitrary sources such as Web services, e-mail, and domain name services.

DETECTING ATTACKS

The detection of an attack is usually through an intrusion detection system. Such systems work by comparing network traffic patterns to a database. In the case of misuse detection, the traffic pattern is compared to historic patterns of known attacks. In the case of anomaly detection, the traffic pattern is compared to a historical baseline of itself. Frequently, the packets must be filtered in order to make comparisons. Filtering can be on the basis of protocol, TCP flags, payload sizes, source or destination address, or port number.

Intrusion detectors must have some sort of sensor to detect attacks, managers to do sensor fusion, databases for storing events for later analysis, tools for offline reporting and analysis, and a control console so that the analyst can modify intrusion detection actions.

RECOVERING FROM ATTACKS

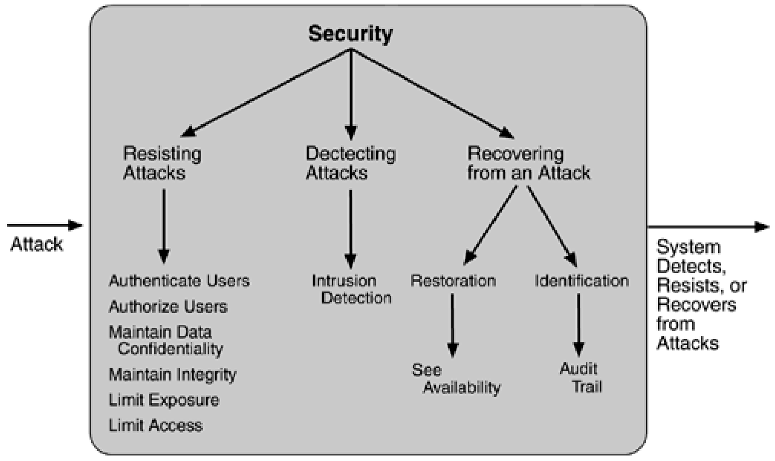
Tactics involved in recovering from an attack can be divided into those concerned with restoring state and those concerned with attacker identification (for either preventive or punitive purposes).

The tactics used in restoring the system or data to a correct state overlap with those used for availability since they are both concerned with recovering a consistent state from an inconsistent state. One difference is that special attention is paid to maintaining redundant copies of system administrative data such as passwords, access control lists, domain name services, and user profile data.

The tactic for identifying an attacker is to maintain an audit trail. An audit trail is a copy of each transaction applied to the data in the system together with identifying information. Audit information can be used to trace the actions of an attacker, support nonrepudiation (it provides evidence that a particular request was made), and support system recovery. Audit trails are often attack targets themselves and therefore should be maintained in a trusted fashion.

Figure 5.9 provides a summary of the tactics for security.

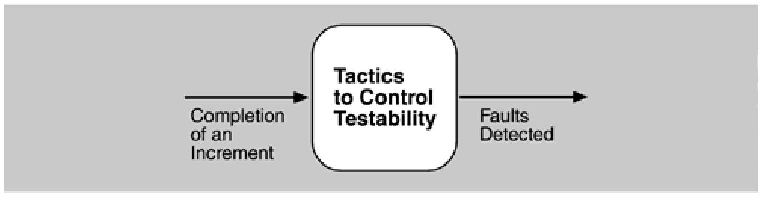
Figure 5.9. Summary of tactics for security



5.6. Testability Tactics

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. Figure 5.10 displays the use of tactics for testability. Architectural techniques for enhancing the software testability have not received as much attention as more mature fields such as modifiability, performance, and availability, but, as we stated in Chapter 4, since testing consumes such a high percentage of system development cost, anything the architect can do to reduce this cost will yield a significant benefit.

Figure 5.10. Goal of testability tactics



Although in Chapter 4 we included design reviews as a testing technique, in this chapter we are concerned only with testing a running system. The goal of a testing regimen is to discover faults. This requires that input be provided to the software being tested and that the output be captured.

Executing the test procedures requires some software to provide input to the software being tested and to capture the output. This is called a test harness. A question we do not consider here is the design and generation of the test harness. In some systems, this takes substantial time and expense.

We discuss two categories of tactics for testing: providing input and capturing output, and internal monitoring.

INPUT/OUTPUT

There are three tactics for managing input and output for testing.

- Record/playback. Record/playback refers to both capturing information crossing an interface and using it as input into the test harness. The information crossing an interface during normal operation is saved in some repository and represents output from one component and input to another. Recording this information allows test input for one of the components to be generated and test output for later comparison to be saved.
- Separate interface from implementation. Separating the interface from the implementation allows substitution of implementations for various testing purposes. Stubbing implementations allows the remainder of the system to be tested in the absence of the component being stubbed. Substituting a specialized component allows the component being replaced to act as a test harness for the remainder of the system.
- Specialize access routes/interfaces. Having specialized testing interfaces allows the capturing or specification of variable values for a component through a test harness as well as independently from its normal execution. For example, metadata might be made available through a specialized interface that a test harness would use to drive its activities. Specialized access routes and interfaces should be kept separate from the access routes and interfaces for required functionality. Having a hierarchy of test interfaces in the architecture means that test cases can be applied at any level in the architecture and that the testing functionality is in place to observe the response.

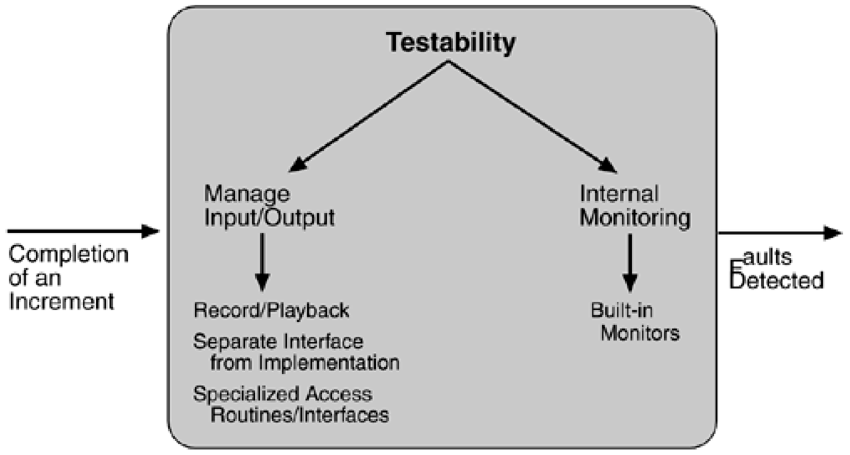
INTERNAL MONITORING

A component can implement tactics based on internal state to support the testing process.

- Built-in monitors. The component can maintain state, performance load, capacity, security, or other information accessible through an interface. This interface can be a permanent interface of the component or it can be introduced temporarily via an instrumentation technique such as aspect-oriented programming or preprocessor macros. A common technique is to record events when monitoring states have been activated. Monitoring states can actually increase the testing effort since tests may have to be repeated with the monitoring turned off. Increased visibility into the activities of the component usually more than outweigh the cost of the additional testing.

Figure 5.11 provides a summary of the tactics used for testability.

Figure 5.11. Summary of testability tactics

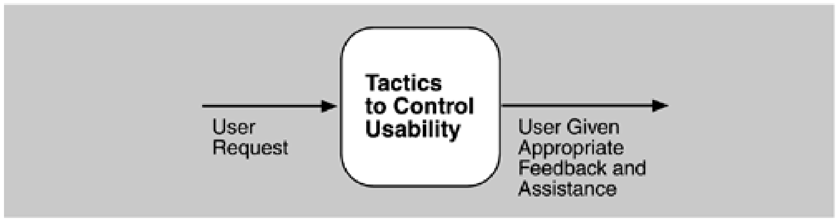

Testability
Manage Input/Output
Internal Monitoring
Completion of an Increment
Record/Playback
Separate Interface from Implementation
Specialized Access Routines/Interfaces
Built-in Monitors
Faults Detected

5.7. Usability Tactics

Recall from Chapter 4 that usability is concerned with how easy it is for the user to accomplish a desired task and the kind of support the system provides to the user. Two types of tactics support usability, each intended for two categories of "users." The first category, runtime, includes those that support the user during system execution. The second category is based on the iterative nature of user interface design and supports the interface developer at design time. It is strongly related to the modifiability tactics already presented.

Figure 5.12 shows the goal of the runtime tactics.

Figure 5.12. Goal of runtime usability tactics


User Request
Tactics to Control Usability
User Given Appropriate Feedback and Assistance

RUNTIME TACTICS

Once a system is executing, usability is enhanced by giving the user feedback as to what the system is doing and by providing the user with the ability to issue usability-based commands such as those we saw in Chapter 4. For example, cancel, undo, aggregate, and show multiple views support the user in either error correction or more efficient operations.

Researchers in human–computer interaction have used the terms "user intiative," "system initiative," and "mixed initiative" to describe which of the human–computer pair takes the initiative in performing certain actions and how the interaction proceeds. The usability scenarios we enumerated in Chapter 4, Understanding Quality Attributes, combine initiatives from both perspectives. For example, when canceling a command the user issues a cancel—"user initiative"—and the system responds. During the cancel, however, the system may put up a progress indicator—"system initiative." Thus, cancel demonstrates "mixed initiative." We use this distinction between user and system initiative to discuss the tactics that the architect uses to achieve the various scenarios.

When the user takes the initiative, the architect designs a response as if for any other piece of functionality. The architect must enumerate the responsibilities of the system to respond to the user command. To use the cancel example again: When the user issues a cancel command, the system must be listening for it (thus, there is the responsibility to have a constant listener that is not blocked by the actions of whatever is being canceled); the command to cancel must be killed; any resources being used by the canceled command must be freed; and components that are collaborating with the canceled command must be informed so that they can also take appropriate action.

When the system takes the initiative, it must rely on some information—a model—about the user, the task being undertaken by the user, or the system state itself. Each model requires various types of input to accomplish its initiative. The system initiative tactics are those that identify the models the system uses to predict either its own behavior or the user's intention. Encapsulating this information will enable an architect to more easily tailor and modify those models. Tailoring and modification can be either dynamically based on past user behavior or offline during development.

- Maintain a model of the task. In this case, the model maintained is that of the task. The task model is used to determine context so the system can have some idea of what the user is attempting and provide various kinds of assistance. For example, knowing that sentences usually start with capital letters would allow an application to correct a lower-case letter in that position.

- Maintain a model of the user. In this case, the model maintained is of the user. It determines the user's knowledge of the system, the user's behavior in terms of expected response time, and other aspects specific to a user or a class of users. For example, maintaining a user model allows the system to pace scrolling so that pages do not fly past faster than they can be read.
- Maintain a model of the system. In this case, the model maintained is that of the system. It determines the expected system behavior so that appropriate feedback can be given to the user. The system model predicts items such as the time needed to complete current activity.

DESIGN-TIME TACTICS

User interfaces are typically revised frequently during the testing process. That is, the usability engineer will give the developers revisions to the current user interface design and the developers will implement them. This leads to a tactic that is a refinement of the modifiability tactic of semantic coherence:

- Separate the user interface from the rest of the application. Localizing expected changes is the rationale for semantic coherence. Since the user interface is expected to change frequently both during the development and after deployment, maintaining the user interface code separately will localize changes to it. The software architecture patterns developed to implement this tactic and to support the modification of the user interface are:

  - Model-View-Controller

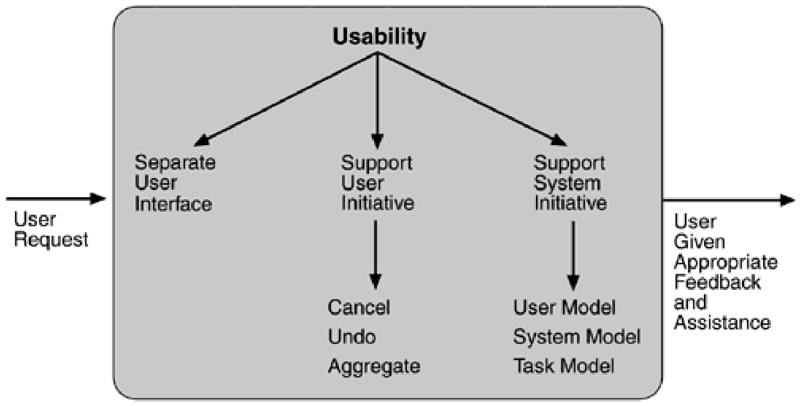
  - Presentation-Abstraction-Control

  - Seeheim

  - Arch/Slinky

Figure 5.13 shows a summary of the runtime tactics to achieve usability.

Figure 5.13. Summary of runtime usability tactics



5.8. Relationship of Tactics to Architectural Patterns

We have presented a collection of tactics that the architect can use to achieve particular attributes. In fact, an architect usually chooses a pattern or a collection of patterns designed to realize one or more tactics. However, each pattern implements multiple tactics, whether desired or not. We illustrate this by discussing the Active Object design pattern, as described by [Schmidt 00]:

The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own thread of control.

The pattern consists of six elements: a proxy, which provides an interface that allows clients to invoke publicly accessible methods on an active object; a method request, which defines an interface for executing the methods of an active object; an activation list, which maintains a buffer of pending method requests; a scheduler, which decides what method requests to execute next; a servant, which defines the behavior and state modeled as an active object; and a future, which allows the client to obtain the result of the method invocation.

The motivation for this pattern is to enhance concurrency—a performance goal. Thus, its main purpose is to implement the "introduce concurrency" performance tactic. Notice the other tactics this pattern involves, however.

- Information hiding (modifiability). Each element chooses the responsibilities it will achieve and hides their achievement behind an interface.
- Intermediary (modifiability). The proxy acts as an intermediary that will buffer changes to the method invocation.
- Binding time (modifiability). The active object pattern assumes that requests for the object arrive at the object at runtime. The binding of the client to the proxy, however, is left open in terms of binding time.
- Scheduling policy (performance). The scheduler implements some scheduling policy.

Any pattern implements several tactics, often concerned with different quality attributes, and any implementation of the pattern also makes choices about tactics. For example, an implementation could maintain a log of requests to the active object for supporting recovery, maintaining an audit trail, or supporting testability.

The analysis process for the architect involves understanding all of the tactics embedded in an implementation, and the design process involves making a judicious choice of what combination of tactics will achieve the system's desired goals.