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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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

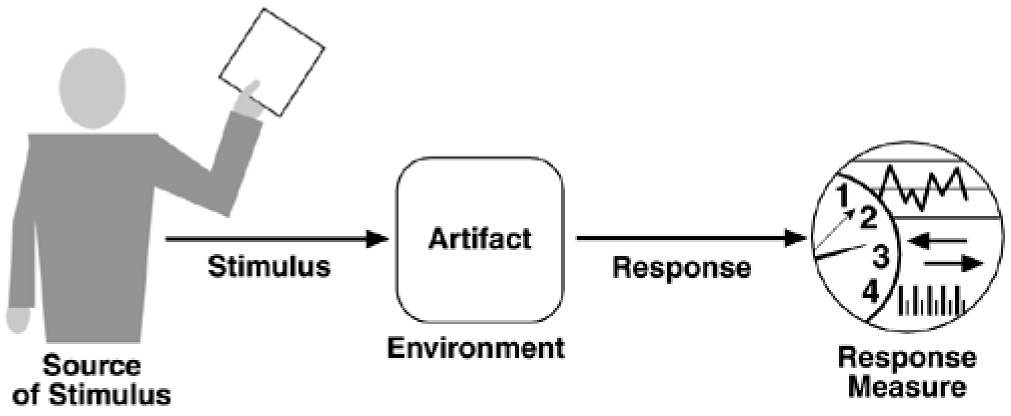
A quality attribute scenario is a quality-attribute-specific requirement. It consists of six parts.

- *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.

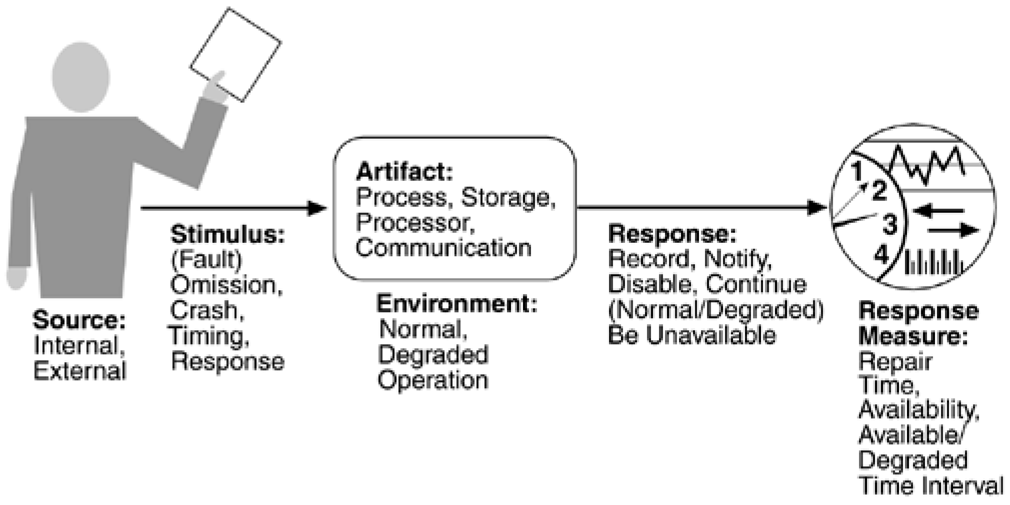
Figure 4.1. Quality attribute parts



Availability Scenario

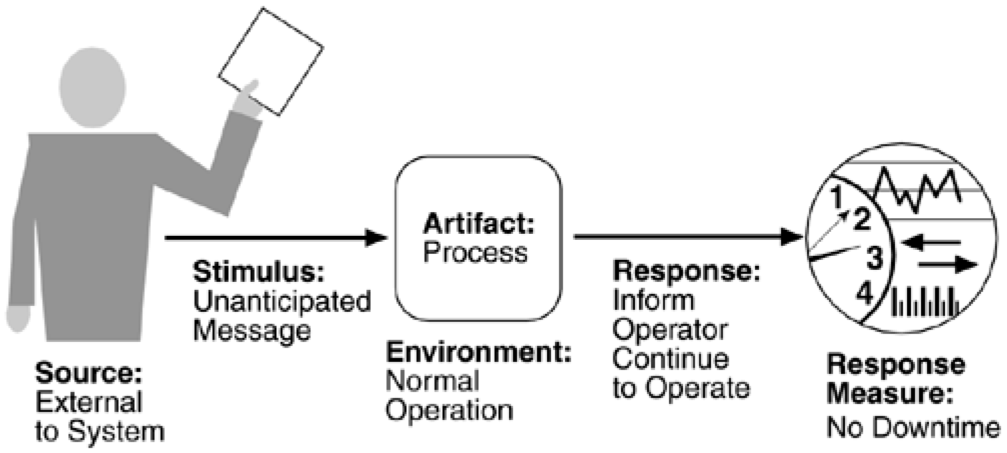
A general scenario for the quality attribute of availability, for example, is shown in Figure 4.2. Its six parts are shown, indicating the range of values they can take. From this we can derive concrete, system-specific, scenarios. Not every system-specific scenario has all of the six parts. The parts 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



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



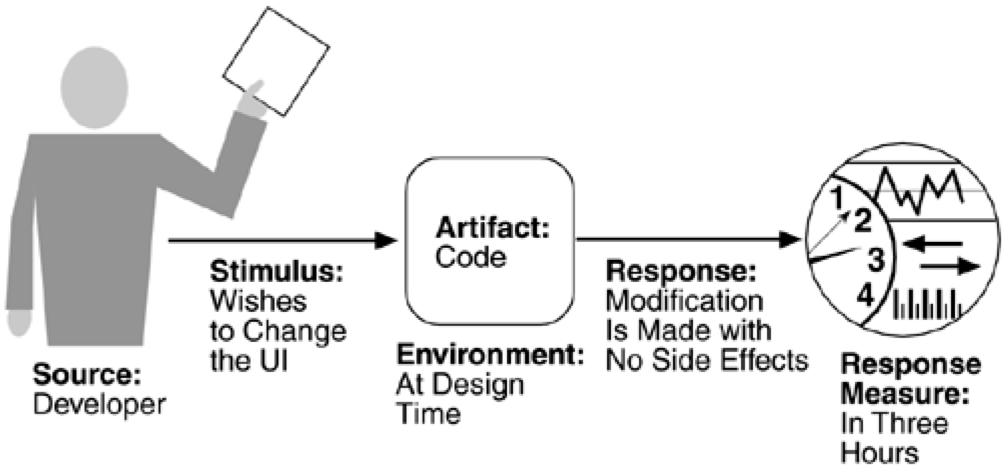
The source of the stimulus is important since differing responses may be required depending on what it is. For example, a request from a trusted source may be treated differently from a request from an untrusted source in a security scenario. The environment may also affect the response, in that an event 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



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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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 Measure	Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or 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.

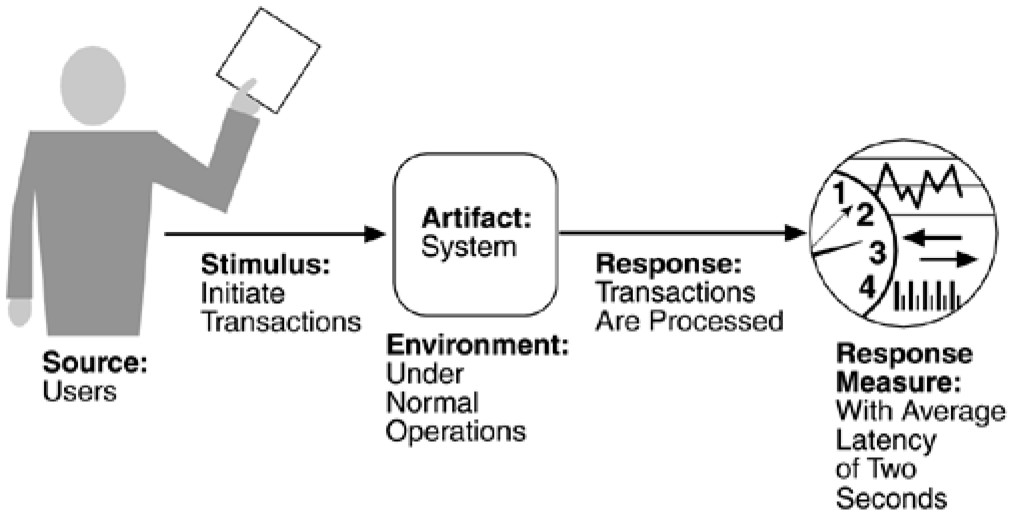
The response of the system to a stimulus can be characterized by latency (the time between the arrival of the stimulus and the system's response to it), deadlines in processing (in the engine controller, for example, the fuel should ignite when the cylinder is in a particular position, thus introducing 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
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.

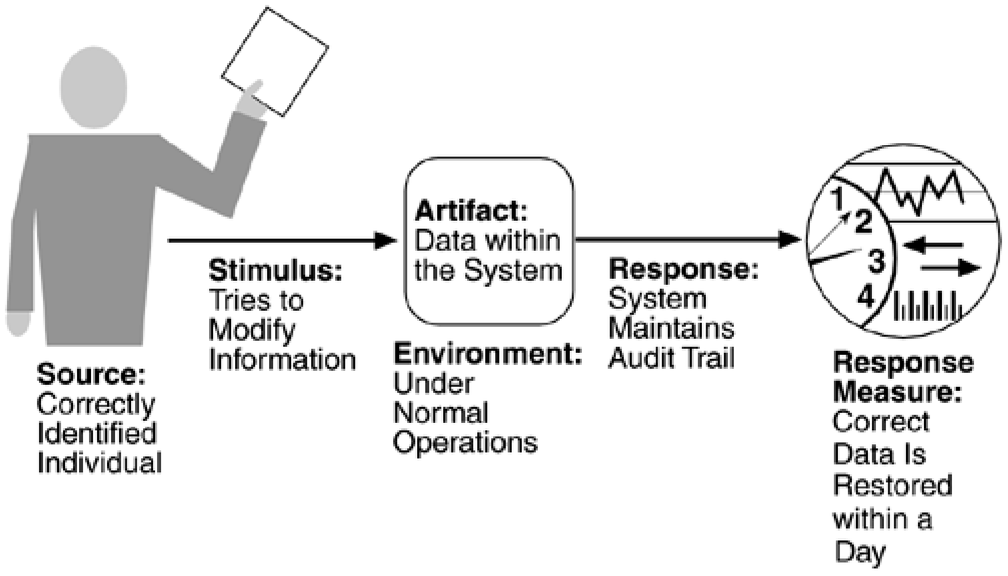
Security General Scenarios

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
1
2
3
4
Response Measure: Correct Data Is Restored within a Day

- *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.

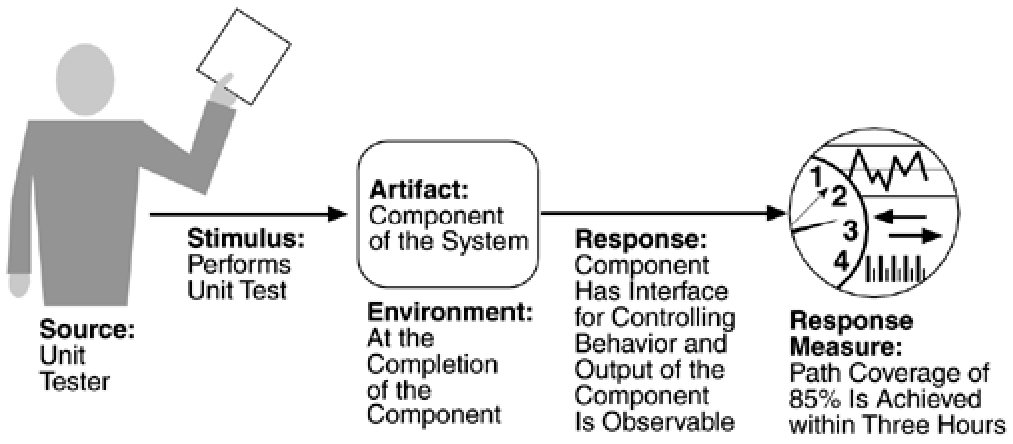
For a system to be properly testable, it must be possible to *control* each component's internal state and inputs and then to *observe* its outputs. Frequently this is done through use of a *test harness*, specialized software designed to exercise the software under test. This may be as simple as a playback 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
1
2
3
4
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
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.

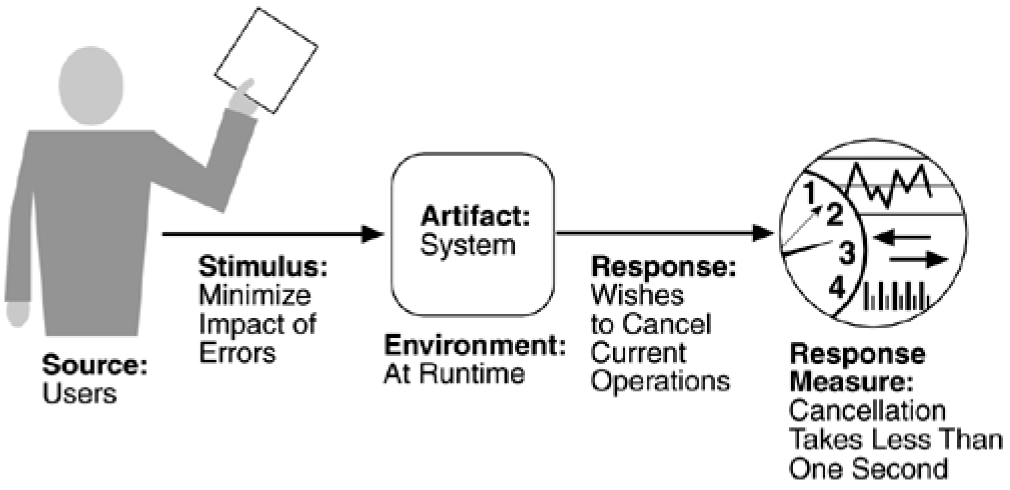
— RK

- *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
1
2
3
4
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
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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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.

[Team LiB] PREVIOUS NEXT
[Team LiB] PREVIOUS NEXT

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.

[Team LiB] PREVIOUS NEXT

4.9 For Further Reading

A discussion of general scenarios and the mapping of scenarios discovered during architectural evaluations to the general scenarios can be found in [Bass 01b]. Further discussion of availability can be found in [Laprie 89] and [Cristian 93]. Security topics can be found in [Ramachandran 02]. The relationship between usability and software architecture is treated in [Gram 96] and [Bass 01a].

[McGregor 01] discusses testability. [Paulish 02] discusses the percentage of development costs associated with testing.

The IEEE maintains standard definitions for quality attributes [ISO 91]. [Witt 94] discusses desirable qualities of architectures (and architects).

4.10 Discussion Questions

1: For the system you are currently working on, what are the most important qualities? What are the system-specific scenarios that capture these qualities and what are the general scenarios they make concrete?

2: Brooks argues that conceptual integrity is the key to successful systems. Do you agree? Can you think of successful systems that have not had this property? If so, what factors made those systems successful anyway? How do you go about measuring a system to see if it meets Brooks's prescription?

3: Generate scenarios for the business and architecture qualities enumerated in Sections 4.4 and 4.5. Have you captured each quality with your scenarios? Which qualities are difficult to capture with scenarios?