# CHAPTER 4: QUALITY ATTRIBUTES

- **Understanding Quality Attributes**
- **Achieving Qualities (Tactics)**

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. 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. We will discuss how the qualities are supported by architectural design decisions, and 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.
- 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.

- **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[refer-Software architecture in practice-Book]. In spite of having the ideal architecture, however, it is always possible to make a system difficult to modify by writing obscure code.
- 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.
- **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).
- 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.
- **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

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. 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.

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

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

- 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.
- **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.

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.

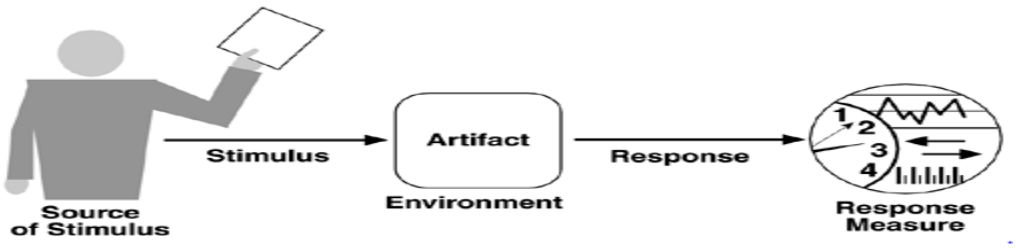


Figure 4.1. Quality attribute parts

### 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.

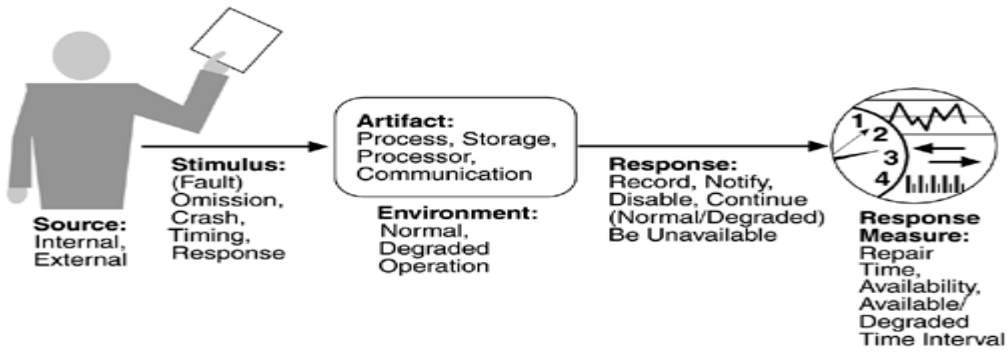


Figure 4.2. Availability general scenarios

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.

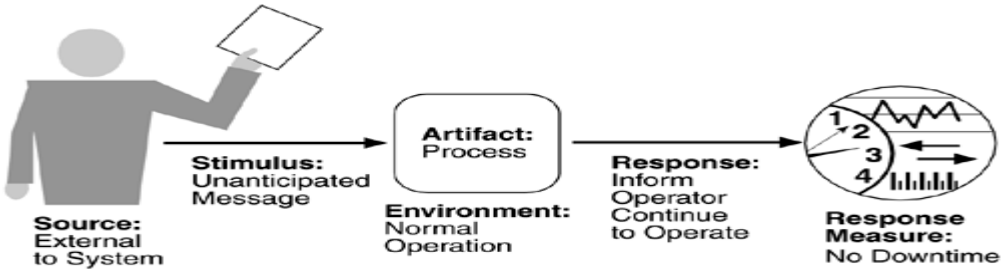


Figure 4.3. Sample availability scenario

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).

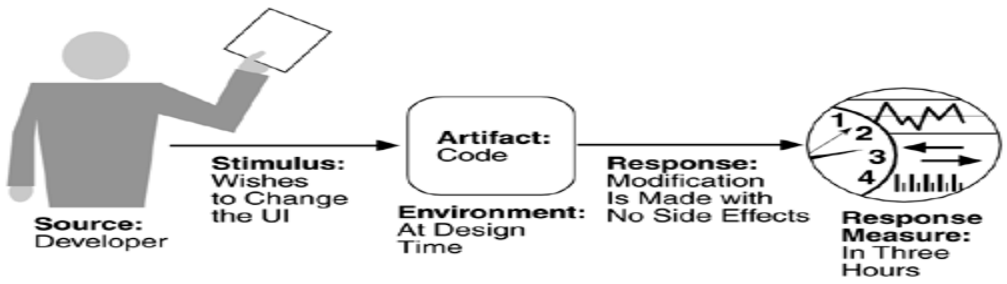


Figure 4.4. Sample modifiability scenario

### 4.5 Performance Scenario

An example of performance scenario 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."

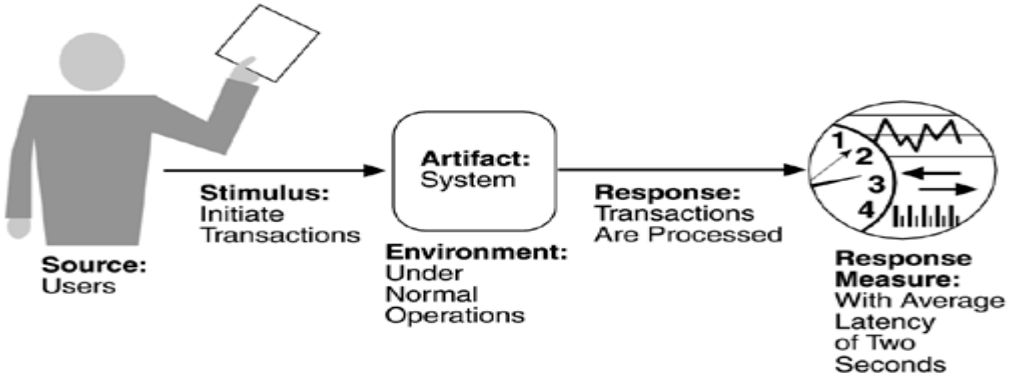


Figure 4.5. Sample performance scenario

### 4.6 Security Scenario

Figure 4.6 presents an example of security scenario. 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.

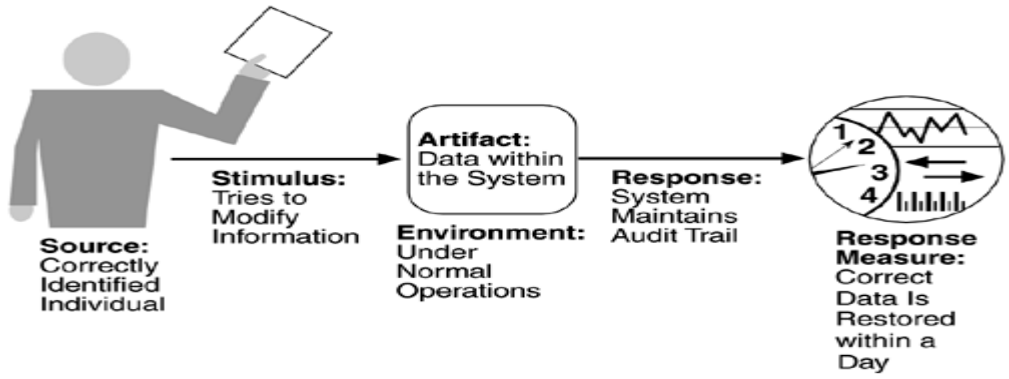


Figure 4.6. Sample security scenario

### 4.7 Testability Scenario

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.

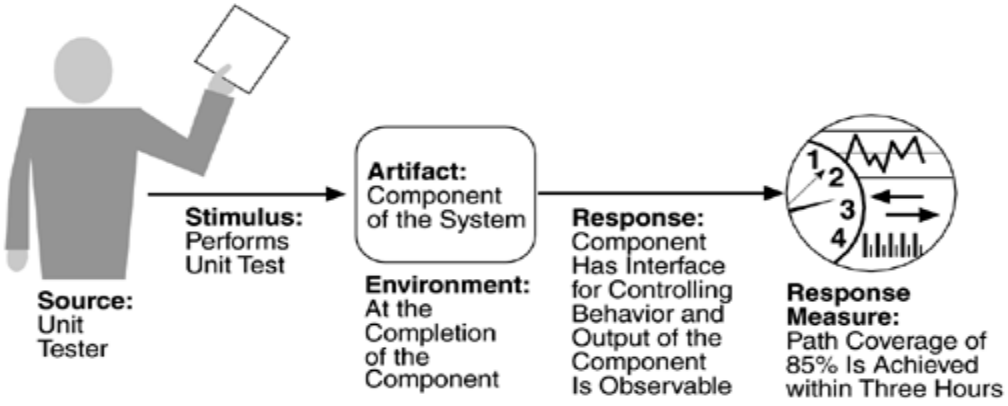


Figure 4.7. Sample testability scenario

### 4.8 Usability Scenario

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.

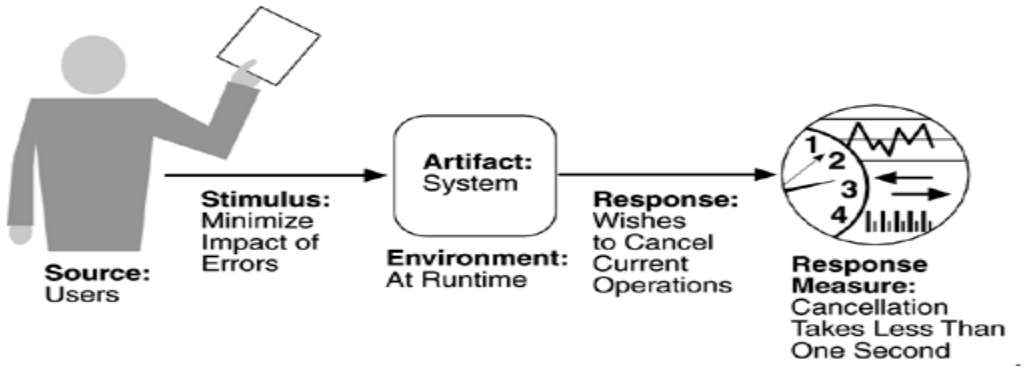


Figure 4.8. Sample usability scenario

## *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 architecture business cycle], 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 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 attributes-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.

### *4.5 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 in house. 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 next chapter, 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.

## 4.6 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 analyzing architectures, 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 analyzing architectures, 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.

## 4.7. Achieving Qualities

We 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 part, we begin to provide that help. For each of the six system quality attributes that we elaborated in previous section, 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.

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 quality attributes) and architectural decisions is the subject of this section.

### 4.7.1 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. An architectural pattern packages tactics in a fashion that we will describe in next section.

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 4.7. 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.

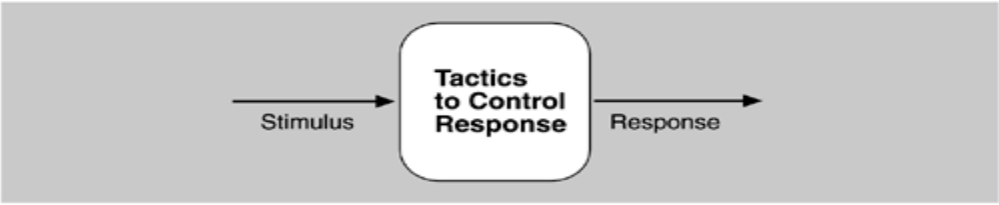


Figure 4.7. Tactics are intended to control responses to stimuli.

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 (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.

### *4.7.2 Availability Tactics*

Recall the vocabulary for availability from previous section. 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 4.7.1.

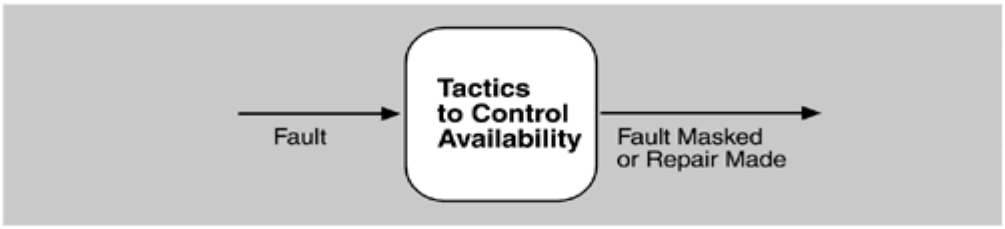


Figure 4.7.1. Goal of availability tactics

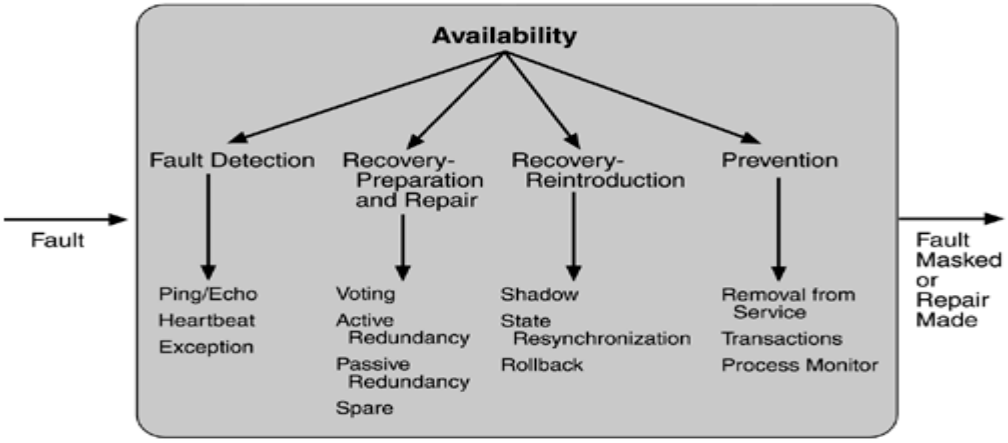


*Figure. Summary of availability tactics*

### *4.7.3 Modifiability Tactics*

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

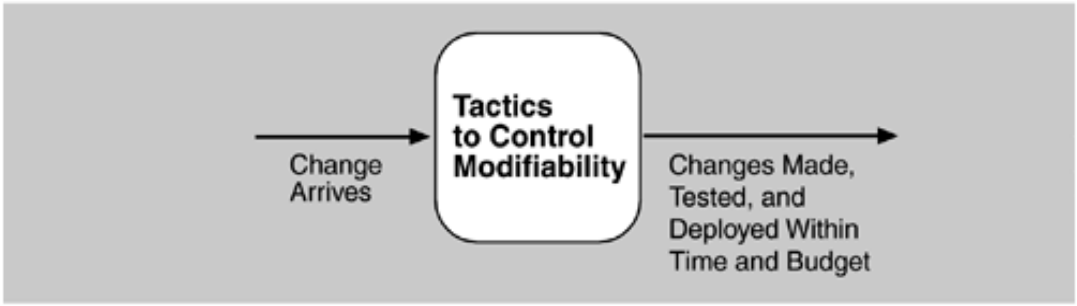


Figure 4.7.2. Goal of modifiability tactics

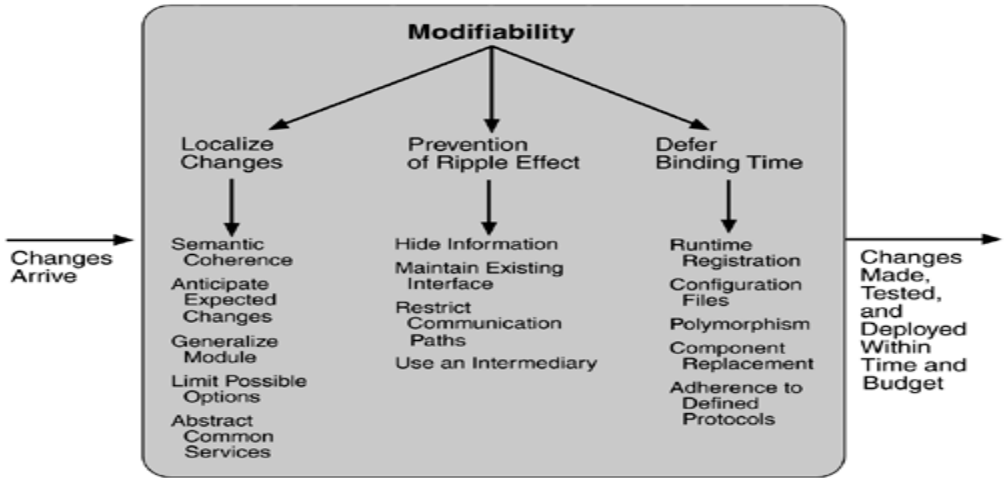


*Figure. Summary of modifiability tactics*

### *4.7.4 Performance Tactics*

Recall from previous 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 4.7.3. Latency is the time between the arrival of an event and the generation of a response to it.

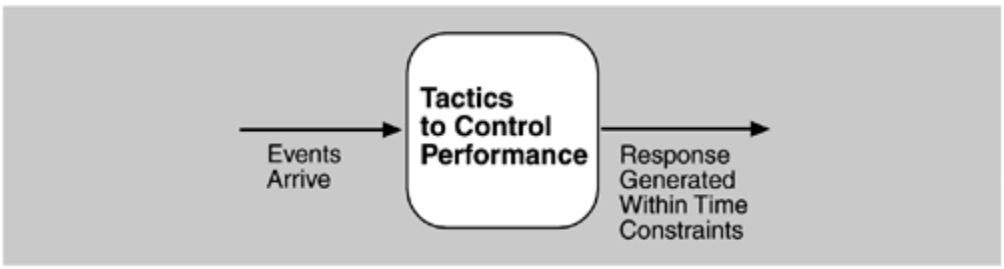


Figure 4.7.3. Goal of performance tactics

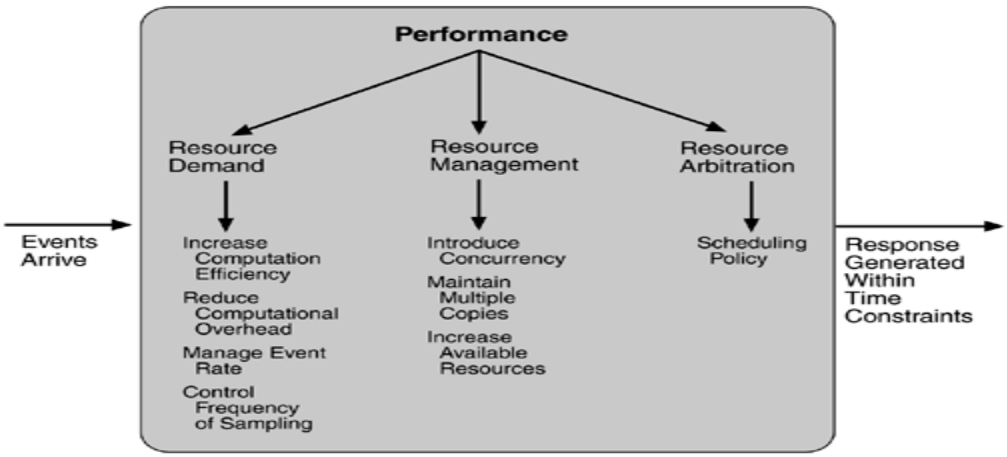


*Figure. Summary of performance tactics*

### *4.7.5 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 4.7.4 shows the goals of the security tactics.

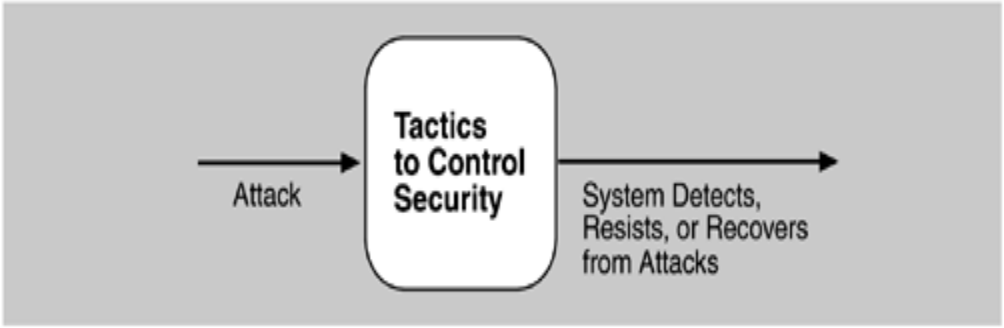


Figure 4.7.4 Goal of security tactics

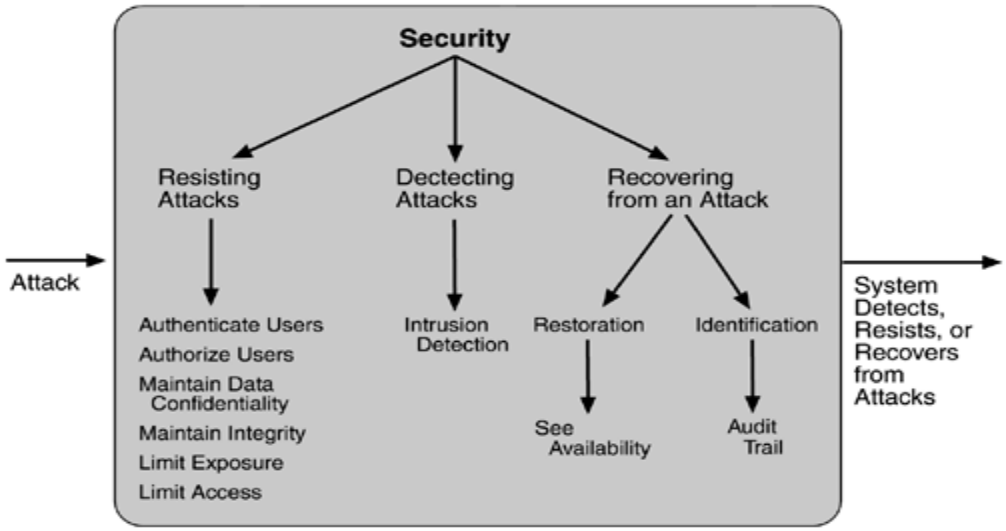


*Figure. Summary of security tactics*

### 4.7.6 Testability Tactics

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. Figure 4.7.5 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 previous, 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.

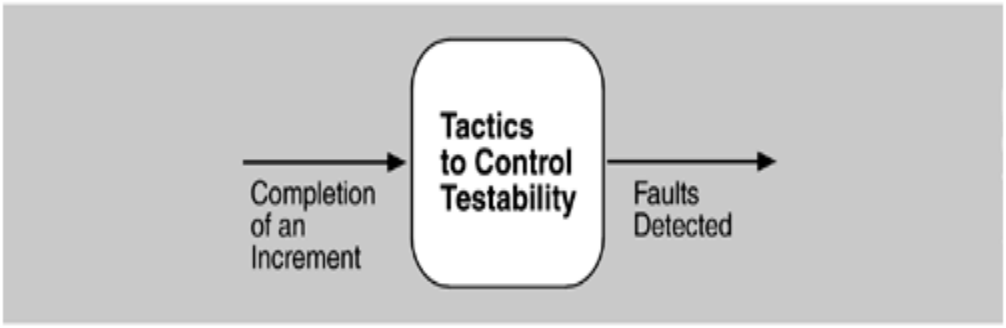


Figure 4.7.5. Goal of testability tactics

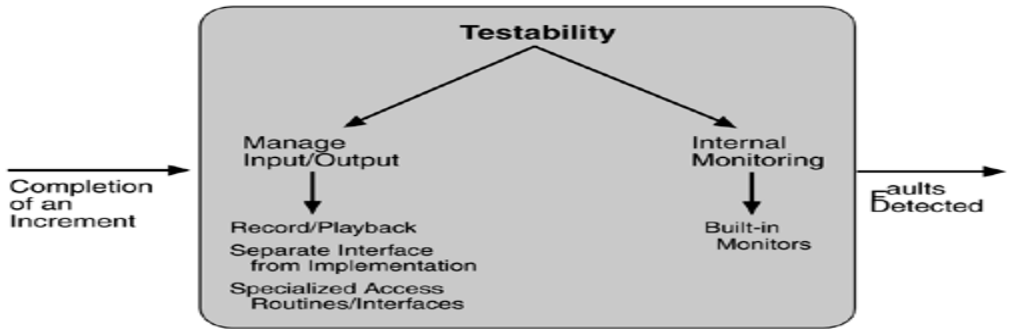


*Figure. Summary of testability tactics*

### *4.7.7 Usability Tactics*

Recall from previous 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 4.7.6 shows the goal of the runtime tactics.

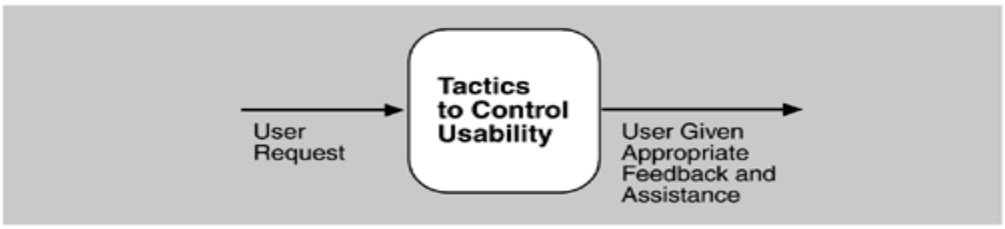


Figure 4.7.6. Goal of runtime usability tactics

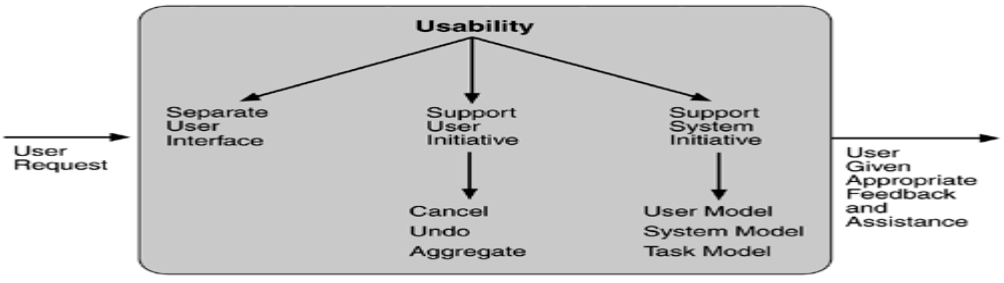


*Figure. Summary of usability tactics*

## 4.8 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.

## 4.9 Architectural Patterns and Styles

An architectural pattern in software, also known as an architectural style, is analogous to an architectural style in buildings, such as Gothic or Greek Revival or Queen Anne. It consists of a

few key features and rules for combining them so that architectural integrity is preserved. An architectural pattern is determined by:

- A set of element types (such as a data repository or a component that computes a mathematical function).
- A topological layout of the elements indicating their interrelation-ships.
- A set of semantic constraints (e.g., filters in a pipe-and-filter style are pure data transducers-they incrementally transform their input stream into an output stream, but do not control either upstream or downstream elements).
- A set of interaction mechanisms (e.g., subroutine call, event-subscriber, blackboard) that determine how the elements coordinate through the allowed topology.

Mary Shaw and David Garlan's influential work attempted to catalog a set of architectural patterns that they called architectural styles or idioms. This has been evolved by the software engineering community into what is now more commonly known as architectural patterns, analogous to design patterns and code patterns.

The motivation of [Shaw 96] for embarking on this project was the observation that high-level abstractions for complex systems exist but we do not study or catalog them, as is common in other engineering disciplines.

These patterns occur not only regularly in system designs but in ways that sometimes prevent us from recognizing them, because in different disciplines the same architectural pattern may be called different things. In response, a number of recurring architectural patterns, their properties, and their benefits have been cataloged. One such catalog is illustrated in Figure 4.9.

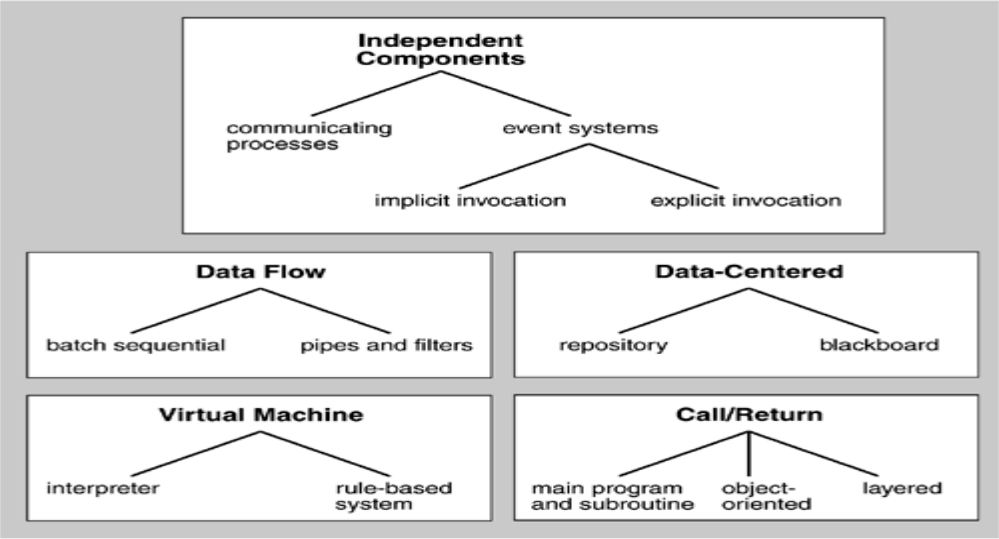


Figure 4.9. A small catalog of architectural patterns, organized by is-a relations

In the above figure patterns are categorized into related groups in an inheritance hierarchy. For example, an event system is a substyle of independent elements. Event systems themselves have two subpatterns: implicit invocation and explicit invocation.

What is the relationship between architectural patterns and tactics? As shown earlier, we view a tactic as a foundational "building block" of design, from which architectural patterns and strategies are created.

### *4.10 Chapter 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 this chapter we also saw how the architect realizes particular quality attribute requirements. These requirements are the means by which a system achieves business goals. Our interest here was in the tactics used by the architect to create a design using architectural patterns and strategies.

We provided a list of well-known tactics for achieving the six quality attributes (availability, modifiability, performance, security, testability, and usability). For each we discussed the tactics that are available and widely practiced.

As we discussed, in relating tactics to patterns the architect's task has only just begun when the tactics are chosen. Any design uses multiple tactics, and understanding what attributes are achieved by them, what their side effects are, and the risks of not choosing other tactics is essential to architecture design.