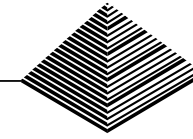
4



Understanding Quality Attributes

Between stimulus and response, there is a space. In that space is our power to choose our response. In our response lies our growth and our freedom.
— Viktor E. Frankl, *Man's Search for Meaning*

As we have seen in the Architecture Influence Cycle (in Chapter 3), many factors determine the qualities that must be provided for in a system's architecture. These qualities go beyond 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 the front seat in the development scheme. This preference is shortsighted, 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 they are too slow; or they have been compromised by hackers. In Chapter 2, we said that architecture was the first place 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 Chapters 5–11 we discuss how various qualities are supported by architectural design decisions. In Chapter 17 we show how to integrate all of the quality attribute decisions into a single design.

We have been using the term "quality attribute" loosely, but now it is time to define it more carefully. A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders. You can think of a quality attribute as measuring the "goodness" of a product along some dimension of interest to a stakeholder.

In this chapter our focus is on understanding the following:

- How to express the qualities we want our architecture to provide to the system or systems we are building from it

- How to achieve those qualities
- How to determine the design decisions we might make with respect to those qualities

This chapter provides the context for the discussion of specific quality attributes in Chapters 5–11.

4.1 Architecture and Requirements

Requirements for a system come in a variety of forms: textual requirements, mockups, existing systems, use cases, user stories, and more. Chapter 16 discusses the concept of an *architecturally significant requirement*, the role such requirements play in architecture, and how to identify them. No matter the source, all requirements encompass the following categories:

1. *Functional requirements.* These requirements state what the system must do, and how it must behave or react to runtime stimuli.
2. *Quality attribute requirements.* These requirements are qualifications of the functional requirements or of the overall product. A qualification of a functional requirement is an item such as how fast the function must be performed, or how resilient it must be to erroneous input. A qualification of the overall product is an item such as the time to deploy the product or a limitation on operational costs.
3. *Constraints.* A constraint is a design decision with zero degrees of freedom. That is, it's a design decision that's already been made. Examples include the requirement to use a certain programming language or to reuse a certain existing module, or a management fiat to make your system service oriented. These choices are arguably in the purview of the architect, but external factors (such as not being able to train the staff in a new language, or having a business agreement with a software supplier, or pushing business goals of service interoperability) have led those in power to dictate these design outcomes.

What is the "response" of architecture to each of these kinds of requirements?

1. Functional requirements are satisfied by assigning an appropriate sequence of responsibilities throughout the design. As we will see later in this chapter, assigning responsibilities to architectural elements is a fundamental architectural design decision.
2. Quality attribute requirements are satisfied by the various structures designed into the architecture, and the behaviors and interactions of the elements that populate those structures. Chapter 17 will show this approach in more detail.

3. Constraints are satisfied by accepting the design decision and reconciling it with other affected design decisions.

4.2 Functionality

Functionality is the ability of the system to do the work for which it was intended. Of all of the requirements, functionality has the strangest relationship to architecture.

First of all, functionality does not determine architecture. That is, given a set of required functionality, there is no end to the architectures you could create to satisfy that functionality. At the very least, you could divide up the functionality in any number of ways and assign the subpieces to different architectural elements.

In fact, if functionality were the only thing that mattered, you wouldn't have to divide the system into pieces at all; a single monolithic blob with no internal structure would do just fine. Instead, we design our systems as structured sets of cooperating architectural elements—modules, layers, classes, services, databases, apps, threads, peers, tiers, and on and on—to make them understandable and to support a variety of other purposes. Those "other purposes" are the other quality attributes that we'll turn our attention to in the remaining sections of this chapter, and the remaining chapters of Part II.

But although functionality is independent of any particular structure, functionality is achieved by assigning responsibilities to architectural elements, resulting in one of the most basic of architectural structures.

Although responsibilities can be allocated arbitrarily to any modules, software architecture constrains this allocation when other quality attributes are important. For example, systems are frequently divided so that several people can cooperatively build them. The architect's interest in functionality is in how it interacts with and constrains other qualities.

4.3 Quality Attribute Considerations

Just as a system's functions do not stand on their own without due consideration of other quality attributes, neither do quality attributes stand on their own; they pertain to the functions of the system. If a functional requirement is "When the user presses the green button, the Options dialog appears," a performance QA annotation might describe how quickly the dialog will appear; an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired; a usability QA annotation might describe how easy it is to learn this function.

Functional Requirements

After more than 15 years of writing and discussing the distinction between functional requirements and quality requirements, the definition of functional requirements still eludes me. Quality attribute requirements are well defined: performance has to do with the timing behavior of the system, modifiability has to do with the ability of the system to support changes in its behavior or other qualities after initial deployment, availability has to do with the ability of the system to survive failures, and so forth.

Function, however, is much more slippery. An international standard (ISO 25010) defines functional suitability as "the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions." That is, functionality is the ability to provide functions. One interpretation of this definition is that functionality describes what the system does and quality describes how well the system does its function. That is, qualities are attributes of the system and function is the purpose of the system.

This distinction breaks down, however, when you consider the nature of some of the "function." If the function of the software is to control engine behavior, how can the function be correctly implemented without considering timing behavior? Is the ability to control access through requiring a user name/password combination not a function even though it is not the purpose of any system?

I like much better the use of the word "responsibility" to describe computations that a system must perform. Questions such as "What are the timing constraints on that set of responsibilities?", "What modifications are anticipated with respect to that set of responsibilities?", and "What class of users is allowed to execute that set of responsibilities?" make sense and are actionable.

The achievement of qualities induces responsibility; think of the user name/password example just mentioned. Further, one can identify responsibilities as being associated with a particular set of requirements.

So does this mean that the term "functional requirement" shouldn't be used? People have an understanding of the term, but when precision is desired, we should talk about sets of specific responsibilities instead.

Paul Clements has long ranted against the careless use of the term "nonfunctional," and now it's my turn to rant against the careless use of the term "functional"—probably equally ineffectually.

—LB

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:

1. The definitions provided for an attribute are not testable. It is meaningless to say that a system will be "modifiable." Every system may be modifiable with respect to one set of changes and not modifiable with respect to another. The other quality attributes are similar in this regard: a system may be robust with respect to some faults and brittle with respect to others. And so forth.
2. Discussion often focuses on which quality a particular concern belongs to. Is a system failure due to a denial-of-service attack an aspect of availability, an aspect of performance, an aspect of security, or an aspect of usability? All four attribute communities would claim ownership of a system failure due to a denial-of-service attack. All are, to some extent, correct. But this doesn't help us, as architects, understand and create architectural solutions to manage the attributes of concern.
3. 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 they are described using different terms.

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

There are two categories of quality attributes on which we focus. The first is those that describe some property of the system at runtime, such as availability, performance, or usability. The second is those that describe some property of the development of the system, such as modifiability or testability.

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, 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. Determining the design that satisfies all of the quality attribute requirements is partially a matter of making the appropriate tradeoffs; we discuss design in Chapter 17. Our purpose here is to provide the context for discussing each quality attribute. In particular, we focus on how quality attributes can be specified, what architectural decisions will enable the achievement of particular quality attributes, and what questions about quality attributes will enable the architect to make the correct design decisions.

4.4 Specifying Quality Attribute Requirements

A quality attribute requirement should be unambiguous and testable. We use a common form to specify all quality attribute requirements. This has the advantage of emphasizing the commonalities among all quality attributes. It has the disadvantage of occasionally being a force-fit for some aspects of quality attributes.

Our common form for quality attribute expression has these parts:

- *Stimulus.* We use the term "stimulus" to describe an event arriving at the system. The stimulus can be an event to the performance community, a user operation to the usability community, or an attack to the security community. We use the same term to describe a motivating action for developmental qualities. Thus, a stimulus for modifiability is a request for a modification; a stimulus for testability is the completion of a phase of development.
- *Stimulus source.* A stimulus must have a source—it must come from somewhere. The source of the stimulus may affect how it is treated by the system. A request from a trusted user will not undergo the same scrutiny as a request by an untrusted user.
- *Response.* How the system should respond to the stimulus must also be specified. The response consists of the responsibilities that the system (for runtime qualities) or the developers (for development-time qualities) should perform in response to the stimulus. For example, in a performance scenario, an event arrives (the stimulus) and the system should process that event and generate a response. In a modifiability scenario, a request for a modification arrives (the stimulus) and the developers should implement the modification—without side effects—and then test and deploy the modification.
- *Response measure.* Determining whether a response is satisfactory—whether the requirement is satisfied—is enabled by providing a response measure. For performance this could be a measure of latency or throughput; for modifiability it could be the labor or wall clock time required to make, test, and deploy the modification.

These four characteristics of a scenario are the heart of our quality attribute specifications. But there are two more characteristics that are important: environment and artifact.

- *Environment.* The environment of a requirement is the set of circumstances in which the scenario takes place. The environment acts as a qualifier on the stimulus. For example, a request for a modification that arrives after the code has been frozen for a release may be treated differently than one that arrives before the freeze. A failure that is the fifth successive failure

of a component may be treated differently than the first failure of that component.

- *Artifact.* Finally, the artifact is the portion of the system to which the requirement applies. Frequently this is the entire system, but occasionally specific portions of the system may be called out. A failure in a data store may be treated differently than a failure in the metadata store. Modifications to the user interface may have faster response times than modifications to the middleware.

To summarize how we specify quality attribute requirements, we capture them formally as six-part scenarios. While it is common to omit one or more of these six parts, particularly in the early stages of thinking about quality attributes, knowing that all parts are there forces the architect to consider whether each part is relevant.

In summary, here are the six parts:

1. *Source of stimulus.* This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. *Stimulus.* The stimulus is a condition that requires a response when it arrives at a system.
3. *Environment.* The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state. For many systems, "normal" operation can refer to one of a number of modes. For these kinds of systems, the environment should specify in which mode the system is executing.
4. *Artifact.* Some artifact is stimulated. This may be a collection of systems, the whole system, or some piece or pieces of it.
5. *Response.* The response is the activity undertaken as the result of the arrival of the stimulus.
6. *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 (which we call "general scenarios" for short)—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 can characterize quality attributes as a collection of general scenarios. Of course, to translate these generic attribute characterizations into requirements for a particular system, the general scenarios need to be made system specific. Detailed examples of these scenarios will be given in Chapters 5–11. Figure 4.1 shows the parts of a quality attribute scenario that we have just discussed. Figure 4.2 shows an example of a general scenario, in this case for availability.

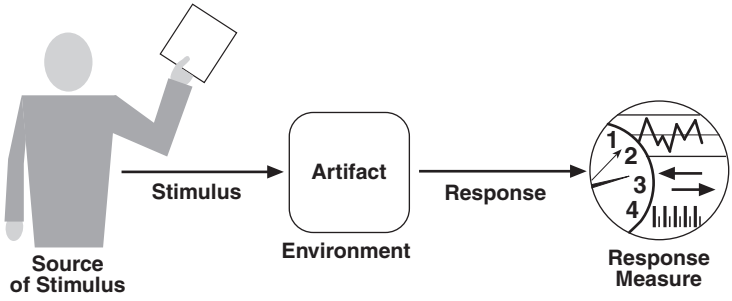


FIGURE 4.1 The parts of a quality attribute scenario

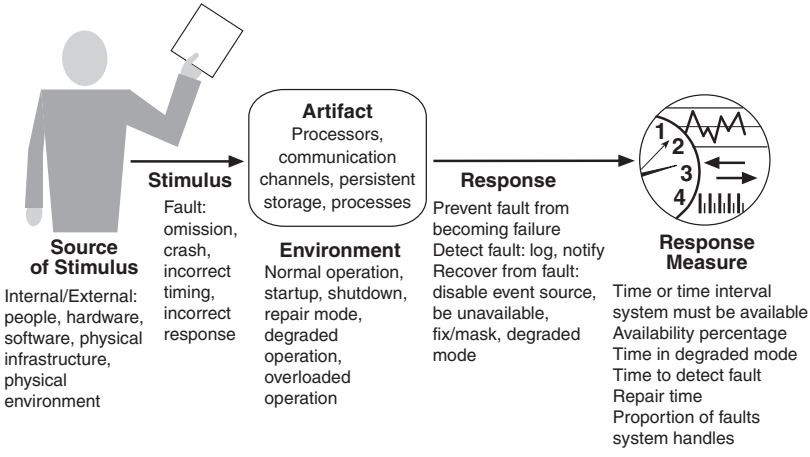


FIGURE 4.2 A general scenario for availability

4.5 Achieving Quality Attributes through Tactics

The quality attribute requirements specify the responses of the system that, with a bit of luck and a dose of good planning, realize the goals of the business. We now turn to the techniques an architect can use to *achieve* the required quality attributes. We call these techniques *architectural tactics.* A tactic is a design decision that influences the achievement of a quality attribute response—tactics directly affect the system's response to some stimulus. Tactics impart portability to one design, high performance to another, and integrability to a third.