CHAPTER 14

Beyond functionality

"Hi, Sam, this is Clarice. I'm presenting a class in the new training room today, but the heating system is terribly loud. I'm practically shouting over the fan and I'm getting hoarse. You're the maintenance supervisor. Why is this system so loud? Is it broken?"

"It's working normally," Sam replied. "The heating system in that room meets the requirements the engineers gave me. It circulates the right amount of air per minute, it controls the temperature to within half a degree from 60 to 85 degrees, and it has all the requested profile programming capabilities. Nobody said anything about noise, so I bought the cheapest system that satisfied the requirements."

Clarice said, "The temperature control is fine. But this is a training room! The students can hardly hear me. We're going to have to install a PA system or get a quieter heating system. What do you suggest?"

Sam wasn't much help. "Clarice, the system meets all the requirements I was given," he repeated. "If I'd known that noise levels were so important, I could have bought a different unit, but now it would be really expensive to replace it. Maybe you can use some throat lozenges so you don't lose your voice."

There's more to software success than just delivering the right functionality. Users also have expectations, often unstated, about *how well* the product will work. Such expectations include how easy it is to use, how quickly it executes, how rarely it fails, how it handles unexpected conditions—and perhaps, how loud it is. Such characteristics, collectively known as *quality attributes, quality factors*, *quality requirements*, *quality of service requirements*, or the "*–ilities*," constitute a major portion of the system's nonfunctional requirements. In fact, to many people, quality attributes are synonymous with nonfunctional requirements, but that's an oversimplification. Two other classes of nonfunctional requirements are constraints (discussed at the end of this chapter) and external interface requirements (discussed in Chapter 10, "Documenting the requirements"). See the sidebar "If they're nonfunctional, then what are they?" in Chapter 1, "The essential software requirement," for more about the term "nonfunctional requirements."

People sometimes get hung up on debating whether a particular need is a functional or a nonfunctional requirement. The categorization matters less than making sure you identify the requirement. This chapter will help you detect and specify nonfunctional requirements you might not have found otherwise.

Quality attributes can distinguish a product that merely does what it's supposed to from one that delights its users. Excellent products reflect an optimum balance of competing quality characteristics.

If you don't explore the customers' quality expectations during elicitation, you're just lucky if the product satisfies them. Disappointed users and frustrated developers are the more typical outcome.

Quality attributes serve as the origin of many functional requirements. They also drive significant architectural and design decisions. It's far more costly to re-architect a completed system to achieve essential quality goals than to design for them at the outset. Consider the many security updates that vendors of operating systems and commonly used applications issue periodically. Some additional work on security at development time might avoid a lot of cost and user inconvenience.

You can't make me

Quality attributes can make or break the success of your product. One large company spent millions of dollars to replace a green-screen call center application with a fancy Windows-based version. After all that investment, the call center representatives refused to adopt the new system because it was too hard to navigate. These power users lost all of the keyboard shortcuts that helped them use the old system efficiently. Now they had to use a mouse to get around in the app, which was slower for them. The corporate leaders first tried the hard-line approach: "We'll just mandate that they have to use the new app," they said. But the call center staff still resisted. What are you going to do? These people are taking customer orders, so the company isn't going to literally turn off the old system if they won't use the new one and risk losing all those orders. Users hate to have their productivity impaired by a "new and improved" system. The development team had to redesign the user interface and add the old keyboard shortcuts before the users would accept the new software, delaying the release by months.

Software quality attributes

Several dozen product characteristics can be called quality attributes, although most project teams need to carefully consider only a handful of them. If developers know which of these characteristics are most crucial to success, they can select appropriate design and construction approaches to achieve the quality goals. Quality attributes have been classified according to a wide variety of schemes (DeGrace and Stahl 1993; IEEE 1998; ISO/IEC 2007; Miller 2009; ISO/IEC 2011). Some authors have constructed extensive hierarchies that group related attributes into several major categories.

One way to classify quality attributes distinguishes those characteristics that are discernible through execution of the software (external quality) from those that are not (internal quality) (Bass, Clements, and Kazman 1998). External quality factors are primarily important to users, whereas internal qualities are more significant to development and maintenance staff. Internal quality attributes indirectly contribute to customer satisfaction by making the product easier to enhance, correct, test, and migrate to new platforms.

Table 14-1 briefly describes several internal and external aspects of quality that every project should consider. Certain attributes are of particular importance on certain types of projects:

- Embedded systems: performance, efficiency, reliability, robustness, safety, security, usability (see Chapter 26, "Embedded and other real-time systems projects")
- Internet and corporate applications: availability, integrity, interoperability, performance, scalability, security, usability
- Desktop and mobile systems: performance, security, usability

In addition, different parts of a system might need to emphasize different quality attributes. Performance could be critical for certain components, with usability being paramount for others. Your environment might have other unique quality attributes that aren't covered here. For example, gaming companies might want to capture emotional requirements for their software (Callele, Neufeld, and Schneider 2008).

Section 6 of the SRS template described in Chapter 10 is devoted to quality attributes. If some quality requirements are specific to certain features, components, functional requirements, or user stories, associate those with the appropriate item in the requirements repository.

TABLE 14-1 Some software quality attributes

External quality	Brief description
Availability	The extent to which the system's services are available when and where they are needed
Installability	How easy it is to correctly install, uninstall, and reinstall the application
Integrity	The extent to which the system protects against data inaccuracy and loss
Interoperability	How easily the system can interconnect and exchange data with other systems or components
Performance	How quickly and predictably the system responds to user inputs or other events
Reliability	How long the system runs before experiencing a failure
Robustness	How well the system responds to unexpected operating conditions
Safety	How well the system protects against injury or damage
Security	How well the system protects against unauthorized access to the application and its data
Usability	How easy it is for people to learn, remember, and use the system
Internal quality	**Brief description**
Efficiency	How efficiently the system uses computer resources
Modifiability	How easy it is to maintain, change, enhance, and restructure the system
Portability	How easily the system can be made to work in other operating environments
Reusability	To what extent components can be used in other systems
Scalability	How easily the system can grow to handle more users, transactions, servers, or other extensions
Verifiability	How readily developers and testers can confirm that the software was implemented correctly

Exploring quality attributes

In an ideal universe, every system would exhibit the maximum possible value for all its attributes. The system would be available at all times, would never fail, would supply instantaneous results that are always correct, would block all attempts at unauthorized access, and would never confuse a user. In reality, there are trade-offs and conflicts between certain attributes that make it impossible to simultaneously maximize all of them. Because perfection is unattainable, you have to determine

which attributes from Table 14-1 are most important to your project's success. Then you can craft specific quality objectives in terms of these essential attributes so designers can make appropriate choices.

Different projects will demand different sets of quality attributes for success. Jim Brosseau (2010) recommends the following practical approach for identifying and specifying the most important attributes for your project. He provides a spreadsheet to assist with the analysis at *www.clarrus.com/resources/articles/software-quality-attributes*.

Step 1: Start with a broad taxonomy

Begin with a rich set of quality attributes to consider, such as those listed in Table 14-1. This broad starting point reduces the likelihood of overlooking an important quality dimension.

Step 2: Reduce the list

Engage a cross-section of stakeholders to assess which of the attributes are likely to be important to the project. (See Figure 2-2 in Chapter 2, "Requirements from the customer's perspective," for an extensive list of possible project stakeholders.) An airport check-in kiosk needs to emphasize usability (because most users will encounter it infrequently) and security (because it has to handle payments). Attributes that don't apply to your project need not be considered further. Record the rationale for deciding that a particular quality attribute is either in or out of consideration.

Recognize, though, that if you don't specify quality goals, no one should be surprised if the product doesn't exhibit the expected characteristics. This is why it's important to get input from multiple stakeholders. In practice, some of the attributes will clearly be in scope, some will clearly be out of scope, and only a few will require discussion about whether they are worth considering for the project.

Step 3: Prioritize the attributes

Prioritizing the pertinent attributes sets the focus for future elicitation discussions. Pairwise ranking comparisons can work efficiently with a small list of items like this. Figure 14-1 illustrates how to use Brosseau's spreadsheet to assess the quality attributes for an airport check-in kiosk. For each cell at the intersection of two attributes, ask yourself, "If I could have only one of these attributes, which would I take?" Entering a less-than sign (<) in the cell indicates that the attribute in the row is more important; a caret symbol (^) points to the attribute at the top of the column as being more important. For instance, comparing availability and integrity, I conclude that integrity is more important. The passenger can always check in with the desk agent if the kiosk isn't operational (albeit, perhaps with a long line of fellow travelers). But if the kiosk doesn't reliably show the correct data, the passenger will be very unhappy. So I put a caret in the cell at the intersection of availability and integrity, pointing up to integrity as being the more important of the two.

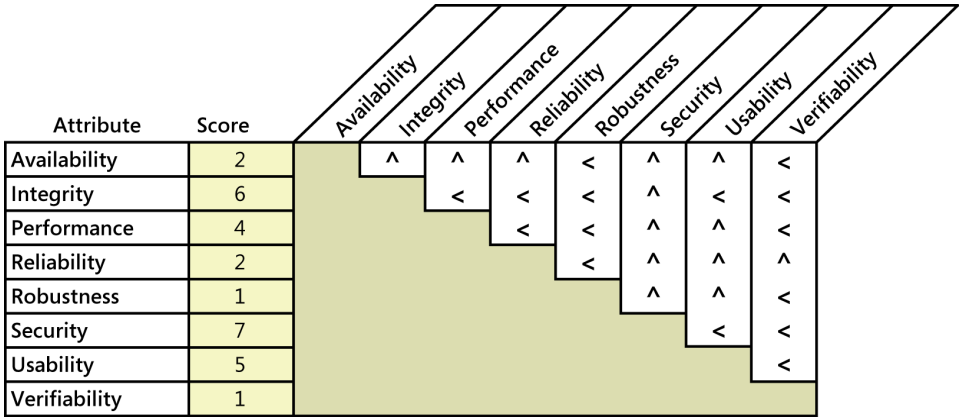

FIGURE 14-1 Sample quality attribute prioritization for an airport check-in kiosk.

The spreadsheet calculates a relative score for each attribute, shown in the second column. In this illustration, security is most important (with a score of 7), closely followed by integrity (6) and usability (5). Though the other factors are indeed important to success—it's not good if the kiosk isn't available for travelers to use or if it crashes halfway through the check-in process—the fact is that not all quality attributes can have top priority.

The prioritization step helps in two ways. First, it lets you focus elicitation efforts on those attributes that are most strongly aligned with project success. Second, it helps you know how to respond when you encounter conflicting quality requirements. In the airport check-in kiosk example, elicitation would reveal a desire to achieve specific performance goals, as well as some specific security goals. These two attributes can clash, because adding security layers can slow down transactions. However, because the prioritization exercise revealed that security is more important (with a score of 7) than performance (with a score of 4), you should bias the resolution of any such conflicts in favor of security.

> **Trap** Don't neglect stakeholders such as maintenance programmers and technical support staff when exploring quality attributes. Their quality priorities could be very different from those of other users. Quality priorities also can vary from one user class to another. If you encounter conflicts, then the approach is doing exactly what it was intended to do: expose these conflicts so you can work through them early in the development life cycle, where conflicts can be resolved with minimal cost and grief.

Step 4: Elicit specific expectations for each attribute

The comments users make during requirements elicitation supply some clues about the quality characteristics they have in mind for the product. The trick is to pin down just what the users are thinking when they say the software must be user-friendly, fast, reliable, or robust. Questions that explore the users' expectations can lead to specific quality requirements that help developers create a delightful product.

Users won't know how to answer questions such as "What are your interoperability requirements?" or "How reliable does the software have to be?" The business analyst will need to ask questions that guide the users' thought processes through an exploration of interoperability, reliability, and other attributes. Roxanne Miller (2009) provides extensive lists of suggested questions to use when eliciting quality requirements; this chapter also presents many examples. When planning an elicitation session, a BA should start with a list of questions like Miller's and distill it down to those questions that are most pertinent to the project. As an illustration, following are a few questions a BA might ask to understand user expectations about the performance of a system that manages applications for patents that inventors have submitted:

1. What would be a reasonable or acceptable response time for retrieval of a typical patent application in response to a query?
2. What would users consider an unacceptable response time for a typical query?
3. How many simultaneous users do you expect on average?
4. What's the maximum number of simultaneous users that you would anticipate?
5. What times of the day, week, month, or year have much heavier usage than usual?

Sending a list of questions like these to elicitation participants in advance gives them an opportunity to think about or research their answers so they don't have to answer a barrage of questions off the tops of their heads. A good final question to ask during any such elicitation discussion is, "Is there anything I haven't asked you that we should discuss?"

Consider asking users what would constitute *unacceptable* performance, security, or reliability. That is, specify system properties that would violate the user's quality expectations, such as allowing an unauthorized user to delete files (Voas 1999). Defining unacceptable characteristics lets you devise tests that try to force the system to demonstrate those characteristics. If you can't force them, you've probably achieved your quality goals. This approach is particularly valuable for safety-critical applications, in which a system that violates reliability or safety tolerances poses a risk to life or limb.

Another possible elicitation strategy is to begin with the quality goals that stakeholders have for the system under development (Alexander and Beus-Dukic 2009). A stakeholder's quality goal can be decomposed to reveal both functional and nonfunctional subgoals—and hence requirements—which become both more specific and easier to measure through the decomposition.

Step 5: Specify well-structured quality requirements

Simplistic quality requirements such as "The system shall be user-friendly" or "The system shall be available 24x7" aren't useful. The former is far too subjective and vague; the latter is rarely realistic or necessary. Neither is measurable. Such requirements provide little guidance to developers. So the final step is to craft specific and verifiable requirements from the information that was elicited regarding each quality attribute. When writing quality requirements, keep in mind the useful SMART mnemonic—make them *Specific*, *Measurable*, *Attainable*, *Relevant*, and *Time-sensitive*.

Quality requirements need to be measurable to establish a precise agreement on expectations among the BA, the customers, and the development team. If it's not measurable, there is little point

in specifying it, because you'll never be able to determine if you've achieved a desired goal. If a tester can't test a requirement, it's not good enough. Indicate the scale or units of measure for each attribute and the target, minimum, and maximum values. The notation called Planguage described later in this chapter helps with this sort of precise specification. It might take a few discussions with users to pin down clear, measurable criteria for assessing satisfaction of a quality requirement.

Suzanne and James Robertson (2013) recommend including *fit criteria*—"a quantification of the requirement that demonstrates the standard the product must reach"—as part of the specification of every requirement, both functional and nonfunctional. This is excellent advice. Fit criteria describe a measurable way to assess whether each requirement has been implemented correctly. They help designers select a solution they believe will meet the goal, and they help testers evaluate the results.

Instead of inventing your own way to document unfamiliar requirements, look for an existing requirement pattern to follow. A pattern provides guidance about how to write a particular type of requirement, along with a template you can populate with the specific details for your situation. Stephen Withall (2007) provides numerous patterns for specifying quality requirements, including performance, availability, flexibility, scalability, security, user access, and installability. Following patterns like these will help even novice BAs write sound quality requirements.

Defining quality requirements

This section describes each of the quality attributes in Table 14-1 and presents some sample quality requirements from various projects. Soren Lauesen (2002) and Roxanne Miller (2009) provide many additional examples of well-specified quality attribute requirements. As with all requirements, it's a good idea to record the origin of each quality requirement and the rationale behind the stated quality goals if these are not obvious. The rationale is important in case questions arise about the need for a specific goal or whether the cost is justifiable. That type of source information has been omitted from the examples presented in this chapter.

External quality attributes

External quality attributes describe characteristics that are observed when the software is executing. They profoundly influence the user experience and the user's perception of system quality. The external quality attributes described in this chapter are availability, installability, integrity, interoperability, performance, reliability, robustness, safety, security, and usability.

Availability

Availability is a measure of the planned up time during which the system's services are available for use and fully operational. Formally, availability equals the ratio of up time to the sum of up time and down time. Still more formally, availability equals the mean time between failures (MTBF) for the system divided by the sum of the MTBF and the mean time to repair (MTTR) the system after a failure is encountered. Scheduled maintenance periods also affect availability. Availability is closely related to reliability and is strongly affected by the maintainability subcategory of modifiability.

Certain tasks are more time-critical than others. Users become frustrated—even irate—when they need to get essential work done and the functionality they need isn't available. Ask users what percentage of up time is really needed or how many hours in a given time period the system must be available. Ask whether there are any time periods for which availability is imperative to meet business or safety objectives. Availability requirements are particularly complex and important for websites, cloud-based applications, and applications that have users distributed throughout many time zones. An availability requirement might be stated like the following:

> *AVL-1. The system shall be at least 95 percent available on weekdays between 6:00 A.M. and midnight Eastern Time, and at least 99 percent available on weekdays between 3:00 P.M. and 5:00 P.M. Eastern Time.*

As with many of the examples presented in this chapter, this requirement is somewhat simplified. It doesn't define the level of performance that constitutes being *available*. Is the system considered available if only one person can use it on the network in a degraded mode? Probably not.

Availability requirements are sometimes stipulated contractually as a service level agreement. Service providers might have to pay a penalty if they do not satisfy such agreements. Such requirements must precisely define exactly what constitutes a system being available (or not) and could include statements such as the following:

> *AVL-2. Down time that is excluded from the calculation of availability consists of maintenance scheduled during the hours from 6:00 P.M. Sunday Pacific Time, through 3:00 A.M. Monday Pacific Time.*

The cost of quality

Beware of specifying 100 percent as the expected value of a quality attribute such as reliability or availability. It will be impossible to achieve and expensive to strive for. Life-critical applications such as air traffic control systems do have very stringent—and legitimate—availability demands. One such system had a "five 9s" requirement, meaning that the system must be available 99.999 percent of the time. That is, the system could be down no more than 5 minutes and 15 seconds per year. This one requirement contributed to perhaps 25 percent of the system costs. It virtually doubled the hardware costs because of the redundancy required, and it introduced very complex architectural elements to handle a hot backup and failover strategy for the system.

When eliciting availability requirements, ask questions to explore the following issues (Miller 2009):

- What portions of the system are most critical for being available?
- What are the business consequences of the system being unavailable to its users?
- If scheduled maintenance must be performed periodically, when should it be scheduled? What is the impact on system availability? What are the minimum and maximum durations of the maintenance periods? How are user access attempts to be managed during the maintenance periods?

- If maintenance or housekeeping activities must be performed while the system is up, what impact will they have on availability and how can that impact be minimized?
- What user notifications are necessary if the system becomes unavailable?
- What portions of the system have more stringent availability requirements than others?
- What availability dependencies exist between functionality groups (such as not accepting credit card payment for purchases if the credit-card authorization function is not available)?

Installability

Software is not useful until it is installed on the appropriate device or platform. Some examples of software installation are: downloading apps to a phone or tablet; moving software from a PC onto a web server; updating an operating system; installing a huge commercial system, such as an enterprise resource planning tool; downloading a firmware update into a cable TV set-top box; and installing an end-user application onto a PC. Installability describes how easy is it to perform these operations correctly. Increasing a system's installability reduces the time, cost, user disruption, error frequency, and skill level needed for an installation operation. Installability addresses the following activities:

- Initial installation
- Recovery from an incomplete, incorrect, or user-aborted installation
- Reinstallation of the same version
- Installation of a new version
- Reverting to a previous version
- Installation of additional components or updates
- Uninstallation

A measure of a system's installability is the mean time to install the system. This depends on a lot of factors, though: how experienced the installer is, how fast the destination computer is, the medium from which the software is being installed (Internet download, local network, CD/DVD), manual steps needed during the installation, and so forth. The Testing Standards Working Party provides a detailed list of guidelines and considerations for installability requirements and installability testing at *www.testingstandards.co.uk/installability_guidelines.htm*. Following are some sample installability requirements:

> *INS-1. An untrained user shall be able to successfully perform an initial installation of the application in an average of 10 minutes.*
>
> *INS-2. When installing an upgraded version of the application, all customizations in the user's profile shall be retained and converted to the new version's data format if needed.*

INS-3. The installation program shall verify the correctness of the download before beginning the installation process.

INS-4. Installing this software on a server requires administrator privileges.

INS-5. Following successful installation, the installation program shall delete all temporary, backup, obsolete, and unneeded files associated with the application.

Following are examples of some questions to explore when eliciting installability requirements:

- What installation operations must be performed without disturbing the user's session?
- What installation operations will require a restart of the application? Of the computer or device?
- What should the application do upon successful, or unsuccessful, installation?
- What operations should be performed to confirm the validity of an installation?
- Does the user need the capability to install, uninstall, reinstall, or repair just selected portions of the application? If so, which portions?
- What other applications need to be shut down before performing the installation?
- What authorization or access privileges does the installer need?
- How should the system handle an incomplete installation, such as one interrupted by a power failure or aborted by the user?

Integrity

Integrity deals with preventing information loss and preserving the correctness of data entered into the system. Integrity requirements have no tolerance for error: the data is either in good shape and protected, or it is not. Data needs to be protected against threats such as accidental loss or corruption, ostensibly identical data sets that do not match, physical damage to storage media, accidental file erasure, or data overwriting by users. Intentional attacks that attempt to deliberately corrupt or steal data are also a risk. Security sometimes is considered a subset of integrity, because some security requirements are intended to prevent access to data by unauthorized users. Integrity requirements should ensure that the data received from other systems matches what is sent and vice versa. Software executables themselves are subject to attack, so their integrity also must be protected.

Data integrity also addresses the accuracy and proper formatting of the data (Miller 2009). This includes concerns such as formatting of fields for dates, restricting fields to the correct data type or length, ensuring that data elements have valid values, checking for an appropriate entry in one field when another field has a certain value, and so on. Following are some sample integrity requirements:

INT-1. After performing a file backup, the system shall verify the backup copy against the original and report any discrepancies.

INT-2. The system shall protect against the unauthorized addition, deletion, or modification of data.

INT-3. The Chemical Tracking System shall confirm that an encoded chemical structure imported from third-party structure-drawing tools represents a valid chemical structure.

INT-4. The system shall confirm daily that the application executables have not been modified by the addition of unauthorized code.

Some factors to consider when discussing integrity requirements include the following (Withall 2007):

- Ensuring that changes in the data are made either entirely or not at all. This might mean backing out of a data change if a failure is encountered partway through the operation.
- Ensuring the persistence of changes that are made in the data.
- Coordinating changes made in multiple data stores, particularly when changes have to be made simultaneously (say, on multiple servers) and at a specific time (say, at 12:00 A.M. GMT on January 1 in several locations).
- Ensuring the physical security of computers and external storage devices.
- Performing data backups. (At what frequency? Automatically and/or on demand? Of what files or databases? To what media? With or without compression and verification?)
- Restoring data from a backup.
- Archiving of data: what data, when to archive, for how long, with what deletion requirements.
- Protecting data stored or backed up in the cloud from people who aren't supposed to access it.

Interoperability

Interoperability indicates how readily the system can exchange data and services with other software systems and how easily it can integrate with external hardware devices. To assess interoperability, you need to know which other applications the users will employ in conjunction with your product and what data they expect to exchange. Users of the Chemical Tracking System were accustomed to drawing chemical structures with several commercial tools, so they presented the following interoperability requirement:

IOP-1. The Chemical Tracking System shall be able to import any valid chemical structure from the ChemDraw (version 13.0 or earlier) and MarvinSketch (version 5.0 or earlier) tools.

You might prefer to state this as an external interface requirement and define the information formats that the Chemical Tracking System can import. You could also define several functional requirements that deal with the import operation. Identifying and documenting such requirements is more important than exactly how you classify them.

Trap Don't store the same requirement in several places, even if it logically fits. That's an invitation to generate an inconsistency if you change, for example, an interoperability requirement but forget to change the same information that you also recorded as a functional or external interface requirement.

Interoperability requirements might dictate that standard data interchange formats be used to facilitate exchanging information with other software systems. Such a requirement for the Chemical Tracking System was:

> *IOP-2. The Chemical Tracking System shall be able to import any chemical structure encoded using the SMILES (simplified molecular-input line-entry system) notation.*

Thinking about the system from the perspective of quality attributes sometimes reveals previously unstated requirements. The users hadn't expressed this chemical structure interoperability need when we were discussing either external interfaces or system functionality. As soon as the BA asked about other systems to which the Chemical Tracking System had to connect, though, the product champion immediately mentioned the two chemical structure drawing packages.

Following are some questions you can use when exploring interoperability requirements:

- To what other systems must this one interface? What services or data must they exchange?
- What standard data formats are necessary for data that needs to be exchanged with other systems?
- What specific hardware components must interconnect with the system?
- What messages or codes must the system receive and process from other systems or devices?
- What standard communication protocols are necessary to enable interoperability?
- What externally mandated interoperability requirements must the system satisfy?

Performance

Performance is one of the quality attributes that users often will bring up spontaneously. Performance represents the responsiveness of the system to various user inquiries and actions, but it encompasses much more than that, as shown in Table 14-2. Withall (2007) provides patterns for specifying several of these classes of performance requirements.

Poor performance is an irritant to the user who's waiting for a query to display results. But performance problems can also represent serious risks to safety, such as when a real-time process control system is overloaded. Stringent performance requirements strongly affect software design strategies and hardware choices, so define performance goals that are appropriate for the operating

environment. All users want their applications to run instantly, but the real performance requirements will be different for a spell-check feature than for a missile's radar guidance system. Satisfying performance requirements can be tricky because they depend so much upon external factors such as the speed of the computer being used, network connections, and other hardware components.

TABLE 14-2 Some aspects of performance

Performance dimension	Example
Response time	Number of seconds to display a webpage
Throughput	Credit card transactions processed per second
Data capacity	Maximum number of records stored in a database
Dynamic capacity	Maximum number of concurrent users of a social media website
Predictability in real-time systems	Hard timing requirements for an airplane's flight-control system
Latency	Time delays in music recording and production software
Behavior in degraded modes or overloaded conditions	A natural disaster leads to a massive number of emergency telephone system calls

When documenting performance requirements, also document their rationale to guide the developers in making appropriate design choices. For instance, stringent database response time demands might lead the designers to mirror the database in multiple geographical locations. Specify the number of transactions per second to be performed, response times, and task scheduling relationships for real-time systems. You could also specify memory and disk space requirements, concurrent user loads, or the maximum number of rows stored in database tables. Users and BAs might not know all this information, so plan to collaborate with various stakeholders to research the more technical aspects of quality requirements. Following are some sample performance requirements:

> *PER-1. Authorization of an ATM withdrawal request shall take no more than 2.0 seconds.*
>
> *PER-2. The anti-lock braking system speed sensors shall report wheel speeds every 2 milliseconds with a variation not to exceed 0.1 millisecond.*
>
> *PER-3. Webpages shall fully download in an average of 3 seconds or less over a 30 megabits/second Internet connection.*
>
> *PER-4. At least 98 percent of the time, the trading system shall update the transaction status display within 1 second after the completion of each trade.*

Performance is an external quality attribute because it can be observed only during program execution. It is closely related to the internal quality attribute of *efficiency*, which has a big impact on the user-observed performance.

Reliability

The probability of the software executing without failure for a specific period of time is known as reliability (Musa 1999). Reliability problems can occur because of improper inputs, errors in the software code itself, components that are not available when needed, and hardware failures. Robustness and availability are closely related to reliability. Ways to specify and measure software reliability include the percentage of operations that are completed correctly, the average length of time the system runs before failing (mean time between failures, or MTBF), and the maximum acceptable probability of a failure during a given time period. Establish quantitative reliability requirements based on how severe the impact would be if a failure occurred and whether the cost of maximizing reliability is justifiable. Systems that require high reliability should also be designed for high verifiability to make it easier to find defects that could compromise reliability.

My team once wrote some software to control laboratory equipment that performed day-long experiments using scarce, expensive chemicals. The users required the software component that actually ran the experiments to be highly reliable. Other system functions, such as logging temperature data periodically, were less critical. A reliability requirement for this system was

> *REL-1. No more than 5 experimental runs out of 1,000 can be lost because of software failures.*

Some system failures are more severe than others. A failure might force the user to re-launch an application and recover data that was saved. This is annoying but not catastrophic. Failures that result in lost or corrupted data, such as when an attempted database transaction fails to commit properly, are more severe. Preventing errors is better than detecting them and attempting to recover from them.

Like many other quality attributes, reliability is a lagging indicator: you can't tell if you've achieved it until the system has been in operation for awhile. Consider the following example:

> *REL-2. The mean time between failures of the card reader component shall be at least 90 days.*

There's no way to tell if the system has satisfied this requirement until at least 90 days have passed. However, you can tell if the system has *failed* to demonstrate sufficient reliability if the card reader component fails more than once within a 90-day period.

Following are some questions to ask user representatives when you're eliciting reliability requirements:

- How would you judge whether this system was reliable enough?
- What would be the consequences of experiencing a failure when performing certain operations with the system?
- What would you consider to be a critical failure, as opposed to a nuisance?
- Under what conditions could a failure have severe repercussions on your business operations?

- No one likes to see a system crash, but are there certain parts of the system that absolutely have to be super-reliable?
- If the system goes down, how long could it stay offline before it significantly affects your business operations?

Understanding reliability requirements lets architects, designers, and developers take actions that they think will achieve the necessary reliability. From a requirements perspective, one way to make a system both reliable and robust is to specify exception conditions and how they are to be handled. Badly handled exceptions can convey an impression of poor reliability and usability to users. A website that blanks out the information a user had entered in a form when it encounters a single bad input value is exasperating. No user would ever specify that behavior as being acceptable. Developers can make systems more reliable by practicing defensive programming techniques, such as testing all input data values for validity and confirming that disk write operations were completed successfully.

Robustness

A customer once told a company that builds measurement devices that its next product should be "built like a tank." The developing company therefore adopted—slightly tongue-in-cheek—the new quality attribute of "tankness." Tankness is a colloquial way of saying *robustness*. Robustness is the degree to which a system continues to function properly when confronted with invalid inputs, defects in connected software or hardware components, external attack, or unexpected operating conditions. Robust software recovers gracefully from problem situations and is forgiving of user mistakes. It recovers from internal failures without adversely affecting the end-user experience. Software errors are handled in a way the user perceives as reasonable, not annoying. Other attribute terms associated with robustness are *fault tolerance* (are user input errors caught and corrected?), *survivability* (can the camera experience a drop from a certain height without damage?), and *recoverability* (can the PC resume proper operation if it loses power in the middle of an operating system update?).

When eliciting robustness requirements, ask users about error conditions the system might encounter and how the system should react. Think about ways to detect possible faults that could lead to a system failure, report them to the user, and recover from them if the failure occurs. Make sure you understand when one operation (such as preparing data for transmission) must be completed correctly before another can begin (sending the data to another computer system). One example of a robustness requirement is

> *ROB-1. If the text editor fails before the user saves the file, it shall recover the contents of the file being edited as of, at most, one minute prior to the failure the next time the same user launches the application.*

A requirement like this might lead a developer to implement checkpointing or periodic autosave to minimize data loss, along with functionality to look for the saved data upon startup and restore the file contents. You wouldn't want to stipulate the precise mechanism in a robustness requirement, though. Leave those technical decisions to developers.

Mea culpa

While writing this chapter, I had a software robustness experience. I was printing a draft chapter and put my computer into sleep mode before the printing was complete, thinking that the data had all been spooled to the printer. It hadn't. How would the print spooler recover from my error when I woke the computer up? Would the spooler terminate and not print the rest of the file, resume printing where it left off, reprint the entire job, or what? It reprinted the entire job, although I would have preferred that it would just continue printing. I wasted some paper, but at least the spooler recovered from my user error and kept going.

I once led a project to develop a reusable software component called the Graphics Engine, which interpreted data files that defined graphical plots and rendered the plots on a designated output device. Several applications that needed to generate plots invoked the Graphics Engine. Because the developers had no control over the data that these applications fed into the Graphics Engine, robustness was an essential quality. One of our robustness requirements was

> *ROB-2. All plot description parameters shall have default values specified, which the Graphics Engine shall use if a parameter's input data is missing or invalid.*

With this requirement, the program wouldn't crash if, for example, an application requested an unsupported line style. The Graphics Engine would supply the default solid line style and continue executing. This would still constitute a product failure because the end user didn't get the desired output. But designing for robustness reduced the severity of the failure from a program crash to generating an incorrect line style, an example of fault tolerance.

Safety

Safety requirements deal with the need to prevent a system from doing any injury to people or damage to property (Leveson 1995; Hardy 2011). Safety requirements might be dictated by government regulations or other business rules, and legal or certification issues could be associated with satisfying such requirements. Safety requirements frequently are written in the form of conditions or actions the system must not allow to occur.

People are rarely injured by exploding spreadsheets. However, hardware devices controlled by software can certainly pose a risk to life and limb. Even some software-only applications can have unobvious safety requirements. An application to let people order meals from a cafeteria might include a safety requirement like the following:

> *SAF-1. The user shall be able to see a list of all ingredients in any menu items, with ingredients highlighted that are known to cause allergic reactions in more than 0.5 percent of the North American population.*

Web browser capabilities like parental controls that disable access to certain features or URLs could be considered as solutions to either safety or security requirements. It's more common to see safety requirements written for systems that include hardware, such as the following examples:

> *SAF-2. If the reactor vessel's temperature is rising faster than 5°C per minute, the Chemical Reactor Control System shall turn off the heat source and signal a warning to the operator.*
>
> *SAF-3. The therapeutic radiation machine shall allow irradiation only if the proper filter is in place.*
>
> *SAF-4. The system shall terminate any operation within 1 second if the measured tank pressure exceeds 90 percent of the specified maximum pressure.*

When eliciting safety requirements you might need to interview subject matter experts who are very familiar with the operating environment or people who have thought a lot about project risks. Consider asking questions like the following:

- Under what conditions could a human be harmed by the use of this product? How can the system detect those conditions? How should it respond?
- What is the maximum allowed frequency of failures that have the potential to cause injury?
- What failure modes have the potential of causing harm or property damage?
- What operator actions have the potential of inadvertently causing harm or property damage?
- Are there specific modes of operation that pose risks to humans or property?

Security

Security deals with blocking unauthorized access to system functions or data, ensuring that the software is protected from malware attacks, and so on. Security is a major issue with Internet software. Users of e-commerce systems want their credit card information to be secure. Web surfers don't want personal information or a record of the sites they visit to be used inappropriately. Companies want to protect their websites against denial-of-service or hacking attacks. As with integrity requirements, security requirements have no tolerance for error. Following are some considerations to examine when eliciting security requirements:

- User authorization or privilege levels (ordinary user, guest user, administrator) and user access controls (the roles and permissions matrix that was illustrated in Figure 9-2 can be a useful tool)
- User identification and authentication (password construction rules, password change frequency, security questions, forgotten logon name or password procedures, biometric identification, account locking after unsuccessful access attempts, unrecognized computer)
- Data privacy (who can create, see, change, copy, print, and delete what information)
- Deliberate data destruction, corruption, or theft
- Protection against viruses, worms, Trojan horses, spyware, rootkits, and other malware

- Firewall and other network security issues
- Encryption of secure data
- Building audit trails of operations performed and access attempts

Following are some examples of security requirements. It's easy to see how you could design tests to verify that these requirements are correctly implemented.

> *SEC-1. The system shall lock a user's account after four consecutive unsuccessful logon attempts within a period of five minutes.*
>
> *SEC-2. The system shall log all attempts to access secure data by users having insufficient privilege levels.*
>
> *SEC-3. A user shall have to change the temporary password assigned by the security officer to a previously unused password immediately following the first successful logon with the temporary password.*
>
> *SEC-4. A door unlock that results from a successful security badge read shall keep the door unlocked for 8.0 seconds, with a tolerance of 0.5 second.*
>
> *SEC-5. The resident antimalware software shall quarantine any incoming Internet traffic that exhibits characteristics of known or suspected virus signatures.*
>
> *SEC-6. The magnetometer shall detect at least 99.9 percent of prohibited objects, with a false positive rate not to exceed 1 percent.*

Security requirements often originate from business rules, such as corporate security policies, as the following example illustrates:

> *SEC-7. Only users who have Auditor access privileges shall be able to view customer transaction histories.*

Try to avoid writing security requirements with embedded design constraints. Specifying passwords for access control is an example. The real requirement is to restrict access to the system to authorized users; passwords are merely one way (albeit the most common way) to accomplish that objective. Depending on which user authentication method is chosen, this security requirement will lead to specific functional requirements that implement the authentication method.

Following are some questions to explore when eliciting security requirements:

- What sensitive data must be protected from unauthorized access?
- Who is authorized to view sensitive data? Who, specifically, is not authorized?
- Under what business conditions or operational time frames are authorized users allowed to access functionality?
- What checks must be performed to confirm that the user is operating the application in a secure environment?

- How frequently should virus software scan for viruses?
- Is there a specific user authentication method that must be used?

Usability

Usability addresses the myriad factors that constitute what people describe colloquially as *user-friendliness, ease of use*, and *human engineering*. Analysts and developers shouldn't talk about "friendly" software but rather about software that's designed for effective and unobtrusive usage. Usability measures the effort required to prepare input for a system, operate it, and interpret its outputs.

Software usability is a huge topic with a considerable body of literature (for example: Constantine and Lockwood 1999; Nielsen 2000; Lazar 2001; Krug 2006; Johnson 2010). Usability encompasses several subdomains beyond the obvious ease of use, including ease of learning; memorability; error avoidance, handling, and recovery; efficiency of interactions; accessibility; and ergonomics. Conflicts can arise between these categories. For instance, ease of learning can be at odds with ease of use. The actions a designer might take to make it easy for a new or infrequent user to employ the system can be irritating impediments to a power user who knows exactly what he wants to do and craves efficiency. Different features within the same application might also have different usability goals. It might be important to be able to enter data very efficiently, but also to be able to easily figure out how to generate a customized report. Table 14-3 illustrates some of these usability design approaches; you can see the possible conflict if you optimize for one aspect of usability over another inappropriately for specific user classes.

Important The key goal for usability—as well as for other quality attributes—is to balance the usability optimally for the whole spectrum of users, not just for a single community. This might mean that certain users aren't as happy with the result as they'd like to be. User customization options can broaden the application's appeal.

TABLE 14-3 Possible design approaches for ease of learning and ease of use

Ease of learning	Ease of use
Verbose prompts	Keyboard shortcuts
Wizards	Rich, customizable menus and toolbars
Visible menu options	Multiple ways to access the same function
Meaningful, plain-language messages	Autocompletion of entries
Help screens and tooltips	Autocorrection of errors
Similarity to other familiar systems	Macro recording and scripting capabilities
Limited number of options and widgets displayed	Ability to carry over information from a previous transaction
	Automatically fill in form fields
	Command-line interface

As with the other quality attributes, it is possible to measure many aspects of "user-friendliness." Usability indicators include:

- The average time needed for a specific type of user to complete a particular task correctly.
- How many transactions the user can complete correctly in a given time period.
- What percentage of a set of tasks the user can complete correctly without needing help.
- How many errors the user makes when completing a task.
- How many tries it takes the user to accomplish a particular task, like finding a specific function buried somewhere in the menus.
- The delay or wait time when performing a task.
- The number of interactions (mouse clicks, keystrokes, touch-screen gestures) required to get to a piece of information or to accomplish a task.

Just tell me what's wrong

Usability shortcomings can be exasperating. I recently tried to report a problem using a website's feedback form. I received an error message that "no special characters were allowed" but the website did not tell me which characters in my text were causing the problem. Obviously, the software knew what the bad characters were because it detected them. Showing me a generic error message instead of offering precise feedback didn't help me solve the problem. I eventually figured out that the software was objecting to the presence of quotation marks in my message. It never occurred to me that quotation marks would be considered a special character; "special character" is vague and ambiguous. To help developers determine how best to satisfy a user's usability expectations, the BA should write specific usability requirements, and developers should provide precise error feedback whenever possible.

To explore their usability expectations, the business analysts on the Chemical Tracking System asked their product champions questions such as "How many steps would you be willing to go through to request a chemical?" and "How long should it take you to complete a chemical request?" These are simple starting points toward defining the many characteristics that will make the software easy to use. Discussions about usability can lead to measurable goals such as the following:

> *USE-1. A trained user shall be able to submit a request for a chemical from a vendor catalog in an average of three minutes, and in a maximum of five minutes, 95 percent of the time.*

Inquire whether the new system must conform to any user interface standards or conventions, or whether its user interface needs to be consistent with those of other frequently used systems. You might state such a usability requirement in the following way:

> *USE-2. All functions on the File menu shall have shortcut keys defined that use the Control key pressed simultaneously with one other key. Menu commands that also appear in Microsoft Word shall use the same default shortcut keys that Word uses.*

Such consistency of usage can help avoid those frustrating errors that occur when your fingers perform an action by habit that has some different meaning in an application you don't use frequently. Ease-of-learning goals also can be quantified and measured, as the following example indicates:

> *USE-3. 95 percent of chemists who have never used the Chemical Tracking System before shall be able to place a request for a chemical correctly with no more than 15 minutes of orientation.*

Carefully specifying requirements for the diverse dimensions of usability can help designers make the choices that distinguish delighted users from those who use an application with frowns on their faces or, worse, those who refuse to use it at all.

Internal quality attributes

Internal quality attributes are not directly observable during execution of the software. They are properties that a developer or maintainer perceives while looking at the design or code to modify it, reuse it, or move it to another platform. Internal attributes can indirectly affect the customer's perception of the product's quality if it later proves difficult to add new functionality or if internal inefficiencies result in performance degradation. The following sections describe quality attributes that are particularly important to software architects, developers, maintainers, and other technical staff.

Efficiency

Efficiency is closely related to the external quality attribute of performance. Efficiency is a measure of how well the system utilizes processor capacity, disk space, memory, or communication bandwidth. If a system consumes too much of the available resources, users will encounter degraded performance.

Efficiency—and hence performance—is a driving factor in systems architecture, influencing how a designer elects to distribute computations and functions across system components. Efficiency requirements can compromise the achievement of other quality attributes. Consider minimum hardware configurations when defining efficiency, capacity, and performance goals. To allow engineering margins for unanticipated conditions and future growth (thereby influencing scalability), you might specify something like the following:

> *EFF-1. At least 30 percent of the processor capacity and memory available to the application shall be unused at the planned peak load conditions.*

EFF-2. The system shall provide the operator with a warning message when the usage load exceeds 80 percent of the maximum planned capacity.

Users won't state efficiency requirements in such technical terms; instead, they will think in terms of response times or other observations. The BA must ask the questions that will surface user expectations regarding issues such as acceptable performance degradation, demand spikes, and anticipated growth. Examples of such questions are:

- What is the maximum number of concurrent users now and anticipated in the future?
- By how much could response times or other performance indicators decrease before users or the business suffer adverse consequences?
- How many operations must the system be able to perform simultaneously under both normal and extreme operating conditions?

Modifiability

Modifiability addresses how easily the software designs and code can be understood, changed, and extended. Modifiability encompasses several other quality attribute terms that relate to different forms of software maintenance, as shown in Table 14-4. It is closely related to verifiability. If developers anticipate making many enhancements, they can choose design approaches that maximize the software's modifiability. High modifiability is critical for systems that will undergo frequent revision, such as those being developed by using an incremental or iterative life cycle.

TABLE 14-4 Some aspects of modifiability

Maintenance type	Modifiability dimensions	Description
Corrective	Maintainability, understandability	Correcting defects
Perfective	Flexibility, extensibility, and augmentability	Enhancing and modifying functionality to meet new business needs and requirements
Adaptive	Maintainability	Modifying the system to function in an altered operating environment without adding new capabilities
Field support	Supportability	Correcting faults, servicing devices, or repairing devices in their operating environment

Ways to measure modifiability include the average time required to add a capability or fix a problem, and the percentage of fixes that are made correctly. The Chemical Tracking System included the following modifiability requirement:

MOD-1. A maintenance programmer experienced with the system shall be able to modify existing reports to conform to revised chemical-reporting regulations from the federal government with 10 hours or less of development effort.

On the Graphics Engine project, we knew we would be doing frequent software surgery to satisfy evolving user needs. Being experienced developers ourselves, we adopted design guidelines such as the following to guide developers in writing the code to enhance the program's understandability and hence maintainability:

> *MOD-2. Function calls shall not be nested more than two levels deep.*

Such design guidelines should be stated carefully to discourage developers from taking silly actions that conform to the letter, but not the intent, of the goal. The BA should work with maintenance programmers to understand what properties of the code would make it easy for them to modify it or correct defects.

Hardware devices containing embedded software often have requirements for supportability in the field. Some of these lead to software design choices, whereas others influence the hardware design. The following is an example of the latter:

> *SUP-1. A certified repair technician shall be able to replace the scanner module in no more than 10 minutes.*

Supportability requirements might also help make the user's life easier, as this example illustrates:

> *SUP-2. The printer shall display an error message if replacement ink cartridges were not inserted in the proper slots.*

Portability

The effort needed to migrate software from one operating environment to another is a measure of portability. Some practitioners include the ability to internationalize and localize a product under the heading of portability. The design approaches that make software portable are similar to those that make it reusable. Portability has become increasingly important as applications must run in multiple environments, such as Windows, Mac, and Linux; iOS and Android; and PCs, tablets, and phones. Data portability requirements are also important.

Portability goals should identify those portions of the product that must be movable to other environments and describe those target environments. One product for analyzing chemicals ran in two very different environments. One version ran in a laboratory where a PhD chemist used the software to control several analytical instruments. The second version ran in a handheld device to be used in the field, such as at an oil pipeline, by someone who had much less technical education. The core capabilities of the two versions were largely the same. Such a product needs to be designed from the outset to work in both kinds of environments with the minimum amount of development work. If developers know about the customers' expectations of portability, they can select development approaches that will enhance the product's portability appropriately. Following are some sample portability requirements:

POR-1. Modifying the iOS version of the application to run on Android devices shall require changing no more than 10 percent of the source code.

POR-2. The user shall be able to port browser bookmarks to and from Firefox, Internet Explorer, Opera, Chrome, and Safari.

POR-3. The platform migration tool shall transfer customized user profiles to the new installation with no user action needed.

When you are exploring portability, questions like the following might be helpful:

- What different platforms will this software need to run on, both now and in the future?
- What portions of the product need to be designed for greater portability than other portions?
- What data files, program components, or other elements of the system need to be portable?
- By making the software more portable, what other quality attributes might be compromised?

Reusability

Reusability indicates the relative effort required to convert a software component for use in other applications. Reusable software must be modular, well documented, independent of a specific application and operating environment, and somewhat generic in capability. Numerous project artifacts offer the potential for reuse, including requirements, architectures, designs, code, tests, business rules, data models, user class descriptions, stakeholder profiles, and glossary terms (see Chapter 18, "Requirements reuse"). Making software reusable is facilitated by thorough specification of requirements and designs, rigorous adherence to coding standards, a maintained regression suite of test cases, and a maintained standard library of reusable components.

Reusability goals are difficult to quantify. Specify which elements of the new system need to be constructed in a manner that facilitates their reuse, or stipulate the reusable components that should be created as a spin-off from the project. Following are some examples:

REU-1. The chemical structure input functions shall be reusable at the object code level in other applications.

REU-2. At least 30 percent of the application architecture shall be reused from the approved reference architectures.

REU-3. The pricing algorithms shall be reusable by future store-management applications.

Consider discussing the following questions when you are trying to learn about reusability requirements for your project:

- What existing requirements, models, design components, data, or tests could be reused in this application?
- What functionality available in related applications might meet certain requirements for this application?

- What portions of this application offer good potential for being reused elsewhere?
- What special actions should be taken to facilitate making portions of this application reusable?

Scalability

Scalability requirements address the ability of the application to grow to accommodate more users, data, servers, geographic locations, transactions, network traffic, searches, and other services without compromising performance or correctness. Scalability has both hardware and software implications. Scaling up a system could mean acquiring faster computers, adding memory or disk space, adding servers, mirroring databases, or increasing network capacity. Software approaches might include distributing computations onto multiple processors, compressing data, optimizing algorithms, and other performance-tuning techniques. Scalability is related to modifiability and to robustness, because one category of robustness has to do with how the system behaves when capacity limits are approached or exceeded. Following are some examples of scalability requirements:

> *SCA-1. The capacity of the emergency telephone system must be able to be increased from 500 calls per day to 2,500 calls per day within 12 hours.*
>
> *SCA-2. The website shall be able to handle a page-view growth rate of 30 percent per quarter for at least two years without user-perceptible performance degradation.*
>
> *SCA-3. The distribution system shall be able to accommodate up to 20 new warehouse centers.*

The business analyst might not have a good sense of future expansion plans for a specific application. She might need to work with the project sponsor or subject matter experts to get a sense of how much the user base, data volume, or other parameters could grow over time. The following questions could be helpful during those discussions:

- What are your estimates for the number of total and concurrent users the system must be able to handle over the next several months, quarters, or years?
- Can you describe how and why data capacity demands of the system might grow in the future?
- What are the minimum acceptable performance criteria that must be satisfied regardless of the number of users?
- What growth plans are available regarding how many servers, data centers, or individual installations the system might be expected to run on?

No, wait, please don't go!

"Cyber Monday" is a marketing term for the Monday following Thanksgiving every November. It has become a traditional day for consumers to shop at online sales for the holiday season. When this custom took root in the mid-2000s, many e-commerce websites weren't prepared to handle the spikes in traffic and transactions from customers shopping for bargains. Servers crashed, passwords weren't recognized, and purchases took too long to be completed. Many shoppers abandoned the online stores they were trying to access and found someplace else to shop, perhaps never to return. Cybercriminals made out, well, like bandits, as traffic was diverted to their look-alike websites that stole shoppers' personal information.

These problems reveal an intertwined mass of unsatisfied software quality requirements. Because of inadequate scalability, systems experienced reliability problems as websites were overwhelmed with visitors, which led to reduced availability. Better software has a direct impact on a company's financial bottom line.

Verifiability

More narrowly referred to as *testability*, verifiability refers to how well software components or the integrated product can be evaluated to demonstrate whether the system functions as expected. Designing for verifiability is critical if the product has complex algorithms and logic, or if it contains subtle functionality interrelationships. Verifiability is also important if the product will be modified often, because it will undergo frequent regression testing to determine whether the changes damaged any existing functionality. Systems with high verifiability can be tested both effectively and efficiently. Designing software for verifiability means making it easy to place the software into the desired pretest state, to provide the necessary test data, and to observe the result of the test. Here are some examples of verifiability requirements:

> *VER-1. The development environment configuration shall be identical to the test configuration environment to avoid irreproducible testing failures.*
>
> *VER-2. A tester shall be able to configure which execution results are logged during testing.*
>
> *VER-3. The developer shall be able to set the computational module to show the interim results of any specified algorithm group for debugging purposes.*

Because my team and I knew that we'd have to test the Graphics Engine many times while it was repeatedly enhanced, we included the following design guideline to enhance verifiability:

> *VER-4. The maximum cyclomatic complexity of a module shall not exceed 20.*

Cyclomatic complexity is a measure of the number of logic branches in a source code module. Adding more branches and loops to a module makes it harder to understand, to test, and to maintain. The project wasn't going to be a failure if some module had a cyclomatic complexity of 24, but documenting such guidelines helped the developers achieve a desired quality objective.

Defining verifiability requirements can be difficult. Explore questions like the following:

- How can we confirm that specific calculations are giving the expected results?
- Are there any portions of the system that do not yield deterministic outputs, such that it could be difficult to determine if they were working correctly?
- Is it possible to come up with test data sets that have a high probability of revealing any errors in the requirements or in their implementation?
- What reference reports or other outputs can we use to verify that the system is producing its outputs correctly?

Specifying quality requirements with Planguage

You can't evaluate a product to judge whether it satisfies vague quality requirements. Unverifiable quality requirements are no better than unverifiable functional requirements. Simplistic quality and performance goals can be unrealistic. Specifying a subsecond response time for a database query might be fine for a simple lookup in a local database but unrealistic for a six-way join of relational tables residing on geographically separated servers.

To address the problem of ambiguous and incomplete nonfunctional requirements, Tom Gilb (1997; 2005) developed *Planguage*, a language with a rich set of keywords that permits precise statements of quality attributes and other project goals (Simmons 2001). Following is an example of how to express a performance requirement using just a few of the many Planguage keywords. Expressed in traditional form, this requirement might read: "At least 95 percent of the time, the system shall take no more than 8 seconds to display any of the predefined accounting reports."

- **TAG** Performance.Report.ResponseTime
- **AMBITION** Fast response time to generate accounting reports on the base user platform.
- **SCALE** Seconds of elapsed time between pressing the Enter key or clicking OK to request a report and the beginning of the display of the report.
- **METER** Stopwatch testing performed on 30 test reports that represent a defined usage operational profile for a field office accountant.
- **GOAL** No more than 8 seconds for 95 percent of reports. ←Field Office Manager
- **STRETCH** No more than 2 seconds for predefined reports, 5 seconds for all reports.
- **WISH** No more than 1.5 seconds for all reports.
- **base user platform DEFINED** Quad-core processor, 8GB RAM, Windows 8, QueryGen 3.3 running, single user, at least 50 percent of system RAM and 70 percent of system CPU capacity free, network connection speed of at least 30 Mbps.

Each requirement receives a unique *tag,* or label, using the hierarchical naming convention that was described in Chapter 10. The *ambition* states the purpose or objective of the system that leads to this requirement. *Scale* defines the units of measurement and *meter* describes how to make the measurements. All stakeholders need to have the same understanding of what "performance" means. Suppose that a user interprets the measurement to be from the time that he presses the Enter key until the complete report appears, rather than until the beginning of the report display, as stated in the example. The developer might claim that the requirement is satisfied, whereas the user insists that it is not. Unambiguous quality requirements and measurements prevent these sorts of debates.

One advantage of Planguage is that you can specify several target values for the quantity being measured. The *goal* criterion is the minimum acceptable achievement level. The requirement isn't satisfied unless every *goal* condition is completely satisfied, so make sure the goals are justifiable in terms of real business needs. An alternative way to state the *goal* requirement is to define the *fail* (another Planguage keyword) condition: "More than 8 seconds on more than 5 percent of all reports." The *stretch* value describes a more desirable performance objective, and the *wish* value represents the ideal outcome. Consider showing the origin of performance goals. The "←" notation following the *goal* criterion shows that it came from the Field Office Manager. Any specialized terms in the Planguage statement are *defined* to make them clear to the reader. This example provides a definition of something called the Base User Platform on which the test is to be conducted.

Planguage includes many additional keywords to provide flexibility and precision in specifying unambiguous quality attribute requirements, and even business objectives. Specifying multiple levels of achievement yields a far richer statement of a quality requirement than a simple black-and-white, yes-or-no construct can. The drawback to using Planguage is that the resulting requirements are much bulkier than simple quality requirement statements. However, the richness of information provided outweighs this inconvenience. Even if you don't write the quality requirements using the full Planguage formalism, using the keywords to think through exactly what people mean by "fast" will yield much more precise and shared expectations.

Quality attribute trade-offs

Certain attribute combinations have inescapable trade-offs. Users and developers must decide which attributes are more important than others, and they must respect those priorities when they make decisions. The technique described earlier in "Step 3: Prioritize the attributes" can help with this analysis. Figure 14-2 illustrates some typical interrelationships among the quality attributes from Table 14-1, although you might encounter exceptions to these (Charette 1990; Glass 1992; IEEE 1998). A plus sign in a cell indicates that increasing the attribute in the corresponding row usually has a positive effect on the attribute in the column. For example, design approaches that increase a software component's portability also make the software easier to connect to other software components, easier to reuse, and easier to test.

	Availability	Efficiency	Installability	Integrity	Interoperability	Modifiability	Performance	Portability	Reliability	Reusability	Robustness	Safety	Scalability	Security	Usability	Verifiability
Availability								+		+						
Efficiency	+				−	−	+	−			−		+		−	
Installability	+							+					+			
Integrity			−		−		−			−		+		+	−	−
Interoperability	+		−	−			−	+	+		+	−		−		
Modifiability	+		−				−		+	+			+			+
Performance		+			−	−		−			−		−		−	
Portability		−			+	−	−			+				−	−	+
Reliability	+	−		+		+	−				+	+		+	+	+
Reusability		−		−	+	+	−	+						−		+
Robustness	+	−	+	+	+		−		+			+	+	+	+	
Safety		−		+	+		−				+			+	−	−
Scalability	+	+		+			+	+	+		+					
Security	+			+	+		−	−	+		+	+			−	−
Usability		−	+				−	−	+		+	+				−
Verifiability	+		+	+		+			+	+	+	+		+	+	

FIGURE 14-2 Positive and negative relationships among selected quality attributes.

A minus sign in a cell means that increasing the attribute in that row generally adversely affects the attribute in the column. An empty cell indicates that the attribute in the row has little effect on the attribute in the column. Performance and efficiency have a negative impact on several other attributes. If you write the tightest, fastest code you can, using coding tricks and relying on execution side effects, it's likely to be hard to maintain and enhance. It also could be harder to port to other platforms if you've tuned the code for a specific operating environment. Similarly, systems that optimize ease of use or that are designed to be reusable and interoperable with other software or hardware components often incur a performance penalty. Using the general-purpose Graphics Engine component described earlier in the chapter to generate plots resulted in poorer performance compared with the old applications that incorporated custom graphics code. You have to balance the possible performance (or other) reductions against the anticipated benefits of your proposed solution to ensure that you're making sensible trade-offs.

The matrix in Figure 14-2 isn't symmetrical because the effect that increasing attribute A has on attribute B isn't necessarily the same as the effect that increasing B will have on A. Figure 14-2 shows that designing the system to increase performance doesn't necessarily have any effect on security. However, increasing security likely will hurt performance because the system must go through more layers of user authentications, encryption, and malware scanning.

To reach the optimum balance of product characteristics, you must identify, specify, and prioritize the pertinent quality attributes during requirements elicitation. As you define the important quality attributes for your project, use Figure 14-2 to avoid making commitments to conflicting goals. Following are some examples:

- Don't expect to maximize usability if the software must run on multiple platforms with minimal modification (portability). Different platforms and operating systems impose different constraints and offer different usability characteristics.
- It's hard to completely test the integrity requirements of highly secure systems. Reused generic components could compromise security mechanisms.
- Highly robust code could exhibit reduced performance because of the data validations and error checking that it performs.

As usual, overconstraining system expectations or defining conflicting requirements makes it impossible for the developers to fully satisfy the requirements.

Implementing quality attribute requirements

Designers and programmers will have to determine the best way to satisfy each quality requirement. Although these are nonfunctional requirements, they can lead to derived functional requirements, design guidelines, or other types of technical information that will produce the desired product characteristics. Table 14-5 indicates the likely categories of technical information that different types of quality attributes will generate. For example, a medical device with stringent availability and reliability requirements might include a backup battery power supply (architecture), along with functional requirements to indicate when the product is operating on battery power, when the battery is getting low, and so forth. This translation from external or internal quality requirements into corresponding technical information is part of the requirements analysis and high-level design processes.

TABLE 14-5 Translating quality attributes into technical specifications

Quality attributes	Likely technical information category
Installability, integrity, interoperability, reliability, robustness, safety, security, usability, verifiability	Functional requirement
Availability, efficiency, modifiability, performance, reliability, scalability	System architecture
Interoperability, security, usability	Design constraint
Efficiency, modifiability, portability, reliability, reusability, scalability, verifiability, usability	Design guideline
Portability	Implementation constraint

Business analysts who lack development experience might not appreciate the technical implications of quality requirements. Therefore, the BA should engage the right stakeholders who have knowledge of these implications and learn from those collaborations. Consider scalability, which

can be profoundly affected by architecture and design choices. Scalability requirements might lead the developer to retain performance buffers (disk space, CPU consumption, network bandwidth) to accommodate potential growth without degrading system performance unacceptably. Scalability expectations can affect the hardware and operating environment decisions that developers make. This is why it's important to elicit and document scalability requirements early on so developers can ensure that the product can grow as expected and still exhibit acceptable performance. This is also one reason why it's important to involve developers early in requirements elicitation and reviews.

Constraints

A constraint places restrictions on the design or implementation choices available to the developer. Constraints can be imposed by external stakeholders, by other systems that interact with the one you're building or maintaining, or by other life cycle activities for your system, such as transition and maintenance. Other constraints result from existing agreements, management decisions, and technical decisions (ISO/IEC/IEEE 2011). Sources of constraints include:

- Specific technologies, tools, languages, and databases that must be used or avoided.
- Restrictions because of the product's operating environment or platform, such as the types and versions of web browsers or operating systems that will be used.
- Required development conventions or standards. (For instance, if the customer's organization will be maintaining the software, the organization might specify design notations and coding standards that a subcontractor must follow.)
- Backward compatibility with earlier products and potential forward compatibility, such as knowing which version of the software was used to create a specific data file.
- Limitations or compliance requirements imposed by regulations or other business rules.
- Hardware limitations such as timing requirements, memory or processor restrictions, size, weight, materials, or cost.
- Physical restrictions because of the operating environment or because of characteristics or limitations of the users.
- Existing interface conventions to be followed when enhancing an existing product.
- Interfaces to other existing systems, such as data formats and communication protocols.
- Restrictions because of the size of the display, as when running on a tablet or phone.
- Standard data interchange formats used, such as XML, or RosettaNet for e-business.

These sorts of constraints often are imposed from external sources and must be respected. Constraints can be imposed inadvertently, though. It's common for users to present "requirements" that are actually solution ideas that describe one particular way the user envisions meeting a need. The BA must detect when a requirement includes a solution idea like this and distinguish

the underlying need from the constraint that the solution imposes. Perhaps the solution the user has in mind is in fact the ideal way to solve the problem, in which case the constraint is perfectly legitimate. More often, the real need is hidden, and the BA must work with the user to articulate the thoughts that led to the presented solution. Asking "why" a few times generally will lead to that real requirement.

Some people say that quality attributes *are* constraints. We prefer to think of certain quality requirements as being the origin of some design or implementation constraints. As Table 14-5 indicated, interoperability and usability requirements are potential sources of design constraints. Portability often imposes implementation constraints to make sure the application can easily be moved from one platform or operating environment to another. For instance, some compilers define an *integer* as being 32 bits long, and others define it as 64 bits. To satisfy a portability requirement, a developer might symbolically define a data type called *WORD* as a 32-bit unsigned integer and use the WORD data type instead of the compiler's default integer data type. This ensures that all compilers will treat data items of type WORD in the same way, which helps to make the system work predictably in different operating environments.

Following are some examples of constraints. You can see how these restrict the options available to the architect, designer, and developer.

> *CON-1. The user clicks at the top of the project list to change the sort sequence. [specific user interface control imposed as a design constraint on a functional requirement]*
>
> *CON-2. Only open source software available under the GNU General Public License may be used to implement the product. [implementation constraint]*
>
> *CON-3. The application must use Microsoft .NET framework 4.5. [architecture constraint]*
>
> *CON-4. ATMs contain only $20 bills. [physical constraint]*
>
> *CON-5. Online payments may be made only through PayPal. [design constraint]*
>
> *CON-6. All textual data used by the application shall be stored in the form of XML files. [data constraint]*

Note that some of these constraints exist to comply with some perhaps-unstated quality expectation. Ask why each constraint is imposed to try to reach that underlying quality requirement. Why must open-source software be used, as stated in CON-2? Perhaps because of a desire for increased modifiability, so that's the requirement that leads to the constraint. Why must a specific version of .NET be used, per CON-3? Perhaps because of an implicit portability or reliability requirement. Remember, a constraint is a perceived solution; asking "why" can lead you to the requirement for which it is thought to be a solution.

Handling quality attributes on agile projects

It can be difficult and expensive to retrofit desired quality characteristics into a product late in development or after delivery. That's why even agile projects that develop requirements and deliver functionality in small increments need to specify significant quality attributes and constraints early in the project. This allows developers to make appropriate architectural and design decisions as a foundation for the desired quality characteristics. Nonfunctional requirements need to have priority alongside user stories; you can't defer their implementation until a later iteration.

It's possible to specify quality attributes in the form of stories:

> *As a help desk technician, I want the knowledge base to respond to queries within five seconds so the customer doesn't get frustrated and hang up.*

However, quality requirements are not implemented in the same discrete way as user stories. They can span multiple stories and multiple iterations. Nor are they always readily divisible into smaller chunks to be implemented across multiple iterations like user stories.

Developers need to keep nonfunctional requirements in mind as they consider the implications of implementing individual user stories. As more functionality is added through a series of iterations, the system's efficiency and hence performance can deteriorate. Specify performance goals and begin performance testing with early iterations, so you can become aware of concerns early enough to take corrective actions.

As you saw in Table 14-5, some quality attributes are the source of derived functionality. On an agile project, quality requirements can spawn new items for the product backlog. Consider the following security requirement:

> *As an account owner, I want to prevent unauthorized users from accessing my account so I don't lose any money.*

This requirement would lead the product owner or business analyst on the project to derive multiple user stories that describe the security-related functionality. These stories can be added to the backlog and planned for implementation in specific iterations in the usual fashion. Understanding these requirements up front ensures that the team implements the security requirements at the right time.

As with user stories, it's possible to write acceptance tests for quality attributes. This is a way to quantify the quality attributes. If a performance goal is stated simply as "The knowledge base must return search results quickly," you can't write tests to define what constitutes "quickly." A better acceptance test would be:

> *Keyword search of the knowledge base takes less than 5 seconds, and preferably less than 3 seconds, to return a result.*

Acceptance tests written in this form can present several acceptable levels of satisfaction for the requirement, much like the Goal, Stretch, and Wish keywords used in Planguage, as discussed earlier in this chapter. You could use the Planguage keywords Scale and Meter to define more precisely what exactly is meant by "return a result" and how to perform the test and evaluate the results.

Part of accepting an iteration as being complete is to assess whether the pertinent nonfunctional requirements are satisfied. Often there is a range of acceptable performance, with some outcomes more desirable than others. As it does for any other software development approach, satisfying quality requirements can distinguish delight from disappointment on agile projects.

Next steps

- Identify several quality attributes from Table 14-1 that might be important to users on your current project. Formulate a few questions about each attribute that will help your users articulate their expectations. Based on the user responses, write one or two specific requirements for each important attribute.
- Examine several documented quality requirements for your project to see if they are verifiable. If not, rewrite them so you could assess whether the expected quality outcomes were achieved in the product.
- Revisit the section titled "Exploring quality attributes" in this chapter and try the spreadsheet approach described to rank-order your important quality attributes. Are the trade-offs between attributes being made on your project in agreement with this priority analysis?
- Rewrite several of the quality attribute examples in this chapter by using Planguage, making assumptions when necessary for the sake of illustration. Can you state those quality requirements with more precision and less ambiguity by using Planguage?
- Examine your users' quality expectations for the system for possible conflicts and resolve them. The favored user classes should have the most influence on making the necessary trade-off choices.
- Trace your quality attribute requirements to the functional requirements, design and implementation constraints, or architectural and design choices that implement them.