Realtime publishers

*The Definitive Guide™ To*

# Quality Application Delivery

*sponsored by*

MICRO FOCUS®
Leading the Evolution™

*Don Jones*

## Copyright Statement

# Chapter 3: Costs of Quality, Barriers to Quality, Benefits of Quality

"Okay," you might be saying, "after two chapters of harping on quality—I'm sold. What'll it cost me?" An excellent question, because quality does indeed cost. In fact, it's easy for companies to get so focused on quality—or particular aspects of it—that they run their development costs up past the point where the quality pays for itself. In other words, contrary as it may seem, you can have *too much quality.*

Take a building as an example, and let's replace the word "quality" with "sturdy;" certainly sturdiness is one thing that people feel points to a quality building. Can you have too much sturdiness? Of course. Traditional residential construction typically uses wood framing, but are there sturdier materials? Definitely. Metal 2 × 4 construction, poured concrete, and other materials offer more strength and longer life, but they do so at a significantly higher cost. So why doesn't everyone simply go with these sturdier materials and techniques? Because in most parts of the country, that extra sturdiness doesn't actually add value to the house. Sure, in a hurricane-prone area, you might be able to easily justify the cost of extra sturdiness by weighing against the damage that hurricane winds will do to a less-sturdy home; in that case, you're offsetting the higher construction costs with demonstrably less damage and a longer structural life. Elsewhere in the country, however, the cost of that extra sturdiness isn't offset by anything, meaning your expense isn't, in the end, worth it.

So it is with software. It's easy to spend a lot of money on some aspect of quality—say, bug-free, or high-performance—but to spend more money than a lower level of quality would have cost you. Therefore, we need to talk about what quality costs—both the lack of quality, and the cost of adding more quality. With that information, we can begin to weigh those costs against one another and strike an appropriate balance.

## Quality Costs—Either Way

Application quality doesn't come in black and white. Instead, it comes in many, many shades of gray. In other words, there's no "total quality" level beyond which no further quality can be achieved. You do, in fact, need to decide "how much quality" is right for you, and to do so, you need to have a firm handle on what that quality will cost, and what it will cost to forgo having that quality. In the next two sections, we'll look at the cost of *not* having quality, and the costs that adding quality imposes on a project; it'll be up to you to determine the correct balance between them for your organization.

## The Cost of No Quality

The only way to measure the cost of not having quality is to examine the symptoms of poor quality, and chart the cost of dealing with those symptoms. To go back to the construction analogy, this is essentially saying that the wind damage suffered by structure type A costs $20,000 per storm to repair. That gives us a hard number against which to compare the cost of a better type of structure—if it'll cost $100,000, then we can expect a payback in five years, and can determine whether that is suitable for us.

In software terms, there are several major areas of poor quality that we can begin to measure. Of course, a guide like this can't offer hard dollar figures; those will vary drastically across different organizations. What we can do here is look at the symptoms and at ways of measuring them so that you can figure out the total dollar value within your environment.

### Bugs

As we've seen in previous chapters, bugs ("defects," if you prefer the polite word) are one of the first symptoms of poor quality that nearly anyone can point to. And they certainly do cost: Major bugs will generate Help desk calls, require troubleshooting and remediation time, require testing of those remediation steps, require deployment, and possibly require additional end-user training. A single major bug can cost an organization tens of thousands of dollars without really trying hard; smaller bugs, however, might simply be categorized as "annoyances" that users can easily work around and that cost the organization very little to live with and to eventually repair.

Any organization that develops software must be prepared to track defects; any tracking system must allow for bugs to be categorized, and that categorization should reflect the cost of the bug to the organization: critical bugs, for example, are ones encountered frequently that are hampering business efforts. Minor bugs are mere annoyances that cost little and can often be worked around or ignored. Any critical bug is going to have some fixed *cost points*:

- Users will continue to report a bug to your Help desk or other support system; they'll also continue to report varying symptoms caused by the same bug; each one costs time (and therefore money) to answer, record, and—in the case of new symptoms of a bug—troubleshoot and classify
- Critical bugs generally cause some loss of production, which equals a loss of money
- Bugs will take time for developers to troubleshoot, resolve, and test
- Releasing new software that resolves a bug will take time and, because no release ever goes perfectly, there will be additional costs for problem resolution, deployment management, and so forth
- The cost of regression testing, which is required to ensure that the "fix" doesn't break anything else

Assume each Help desk call costs about $20 to deal with—a conservative estimate involving only first-tier support (Source: Gartner research in "Help the Help Desk" by John Brandon). Also assume that each bug costs each user about $20 worth of lost productivity—again, a pretty conservative estimate. Assume software developers are paid $91,000 per year (fully loaded, including benefits), testers are paid similarly, and that a critical bug can be resolved and tested by a total of two such individuals over the course of a week. Assume that the bug affects only 1000 users, and that each user encounters the bug an average of 5 times per week (both conservative numbers for a large organization). Assuming that deployment of a fixed application goes smoothly and costs nothing, you're still looking at more than $100,000 in costs ($20 per call multiplied by 1000 users at five times per week) a week for that one bug; if you could have spent anything up to that amount avoiding the bug in the first place, it would have been money well spent.

### Performance

Performance is the one area most organizations will agree that they have significant room to improve. It's hardly rare, for example, to call into a phone support system—whether an airline, your bank, a travel agency, or your insurance company—and have a representative tell you, "the computer is being slow today." A lack of performance in applications seems to be extremely common based solely on anecdotal evidence; what's the cost of this type of poor quality? It can be difficult to measure in some organizations, but easy in others. A travel agency earning a minimum $30 commission per phone call, for example, and employing only 100 phone agents, could earn $24,000 more if each agent answered just one extra call per hour. If each call averaged 10 minutes, you would need to make each call just about 1.3 minutes faster—something you could probably achieve if the computer wasn't "being slow today." Taken across an entire year, that's nearly $9 million in additional commissions, without having to add more phone agents (and their salaries) to your bottom line. Think you could have spent something less than $9 million to improve this aspect of your application's quality? Probably. And yet many businesses don't even consider this type of quality when setting out to develop new software applications!

### User Efficiency

User efficiency is related to performance; rather than the user waiting on the computer, however, the computer is waiting on the user. User efficiency is most often impacted by the application's user interface (UI):

- Is it presenting, and collecting, information in the same order that the information is actually needed or available?
- Do users have to jump back and forth between screens or windows?
- Do they have to re-enter information?

Something as simple as keyboard shortcuts, properly used, can make an enormous difference. A user who can simply hit "tab" to move from input field to input field can be as much as 12% more productive than a user who must repeatedly take their hands from the keyboard and use the mouse to reposition their cursor (Source: Microsoft Corporation study on interface design and software usability, June 2005). If our travel agents from the previous example are using the mouse a lot, then a fairly simple re-engineering of the screen to use "tab" properly will result in that 12% savings might be worth $6 million over the course of a year, simply because the user could move through the application faster, complete calls faster, and move on to the next call faster.

It's really tough to imagine something like a better UI design resulting in that kind of monetary savings or gain. But user efficiency is a well-known factor in other industries, including the home-building analogy we began the chapter with. The rule of thumb for kitchen design, for example, is that the refrigerator, sink, and stove be arranged in a triangle whose sides are not longer than 26 feet in total (this was developed by the Small Homes Council of the School of Architecture at the University of Illinois in the 1950s); by breaking this model, you might cause a homeowner to spend an extra 10 minutes preparing a meal. On a meal requiring one hour to prepare, that's a 16% loss, and over the course of a year, that's more than 60 hours wasted—more than a full work week or a week of paid vacation!

Even minor gains in user efficiency, particularly on repetitive tasks, can result in tremendous savings. For every 10 employees who can realize a mere 10% gain in productivity, you can avoid hiring, or re-task, another employee in that same position (a 10% savings multiplied by 10 employees equals a 100% total savings, which is equal to one full employee). That can really add up.

### Incorrect or Incomplete Output

Another more visible area of poor quality is in applications that produce output that is incomplete or incorrect. Incorrect reports, for example, can lead directly to bad business decisions. For example, suppose an application is incorrectly underreporting the amount of discounts offered to customers, causing the business to decide they can afford to offer even more discounts—creating a direct negative impact to the company's bottom line. Incomplete output can be less obviously dangerous, but in an insidious way, can cost even more as employees spend precious time hunting around to assemble the data they need. If a report simply lacks one piece of data that is readily available *within* the application, it might have cost an additional hour during the design process, an hour during development, and a few hours during testing, to have included that report in the first place—at a cost of maybe $250 or so in salaries (assuming five individuals paid $100,000 each and require an additional hour apiece to design, develop, and test the more-complete report) for the people involved. Yet that incomplete report, if used weekly, might cost two managers an extra 10 minutes in productivity—potentially $860 or so per year (assuming two salaries of $100,000 each).

### Maintenance

Maintenance, like performance, is another area application designers and developers frequently ignore: They're concerned with putting out an application that works, not necessarily one that can be subsequently maintained over the months and years that follow. Yet applications that are not designed to be maintained are often difficult to maintain, and the costs of this poor quality can be terrifically difficult to measure. Is the application storing data that is subject to regulatory or industry compliance or privacy rules? If so, the lack of archiving capability, for example, might result in hundreds of thousands of dollars in fines. Is the application easy to back up and restore? If not, a failure might result in millions of dollars in lost data. And yet most of the technologies that support modern applications, including relational database management systems, make it easy to design and develop such features. Including scheduled archiving capabilities in an application based on Microsoft SQL Server, for example, might require an additional 250 man-hours—less than $32,000 if the application is being developed by a consulting firm charging $125 per hour. If that archiving functionality protects the company from four minor European Union (EU) privacy infractions, it will have paid for itself (the 1998 Data Protection Act specifies a minor-infraction fine equivalent to approximately $8500 per occurrence, including instances where data is improperly retained by companies).

Maintenance can also be related to user efficiency. If an administrator being paid $60,000 per year can save one hour per week on application maintenance (either by making maintenance tasks more efficient or automated), the company saves about $1500 per year—well within the realm of what it might have cost to design automated or more efficient maintenance into the application in the first place.

### Security

Security is an obvious result of quality, and poor security the obvious result of poor quality. What does poor security cost? Unfortunately, in many organizations, nothing; I say "unfortunately" because if there was a clear, consistent cost to poor security, then poor security wouldn't exist. Many organizations, however, have never been affected by a security problem, and have never encountered the tremendous costs associated with it. Think of security as a kind of insurance: It'll cost you something to buy it, and you might never need it. But if you do need it, the insurance is vastly less expensive than the alternative.

Poor security can result in data being disclosed to the wrong people, either to internal users or, worse, to external entities who might be able to misuse the data. Improperly disclosed data might result in direct financial loss, such as fines (in the case of government-protected data, for example, such as patient data in healthcare organizations), or it might result in indirect loss, such as the loss when a competitor gains access to proprietary intellectual property. It might result in significant long-term loss. For example, suppose your company develops a new technique for manufacturing widgets that will result in a $5 million yearly savings, enabling you to slightly undercut your competitor while maintaining your margin. This improvement cost you $10 million to develop, and you expect a complete return on that investment in 3 years. Prior to securing a patent on the process, however, it's accidentally disclosed due to a lack of application security, and your competitor gets hold of it. That kind of disclosure can lead to the patent never being awarded, you and your competitor again being on equal footing—and you eating that $10 million investment. All for want of a little security. This isn't even an extreme example; anecdotal evidence suggests that many companies "leak" a significant percentage of their revenue through relatively minor security problems in their enterprise applications.

### Flexibility (Other Languages, Accessibility)

*Suitability for the business' needs* is perhaps the best generic definition of software quality, and flexibility can certainly contribute to the application's suitability. Is the application accessible to all employees, even those who might not have full use of a mouse or the keyboard, or who might have minor visual impairments such as color blindness? Can the application be easily localized into other languages so that the company's international users can operate it in their native language? Building these capabilities into an application isn't exceedingly minor, but it isn't exceedingly difficult, either; modern software development tools and frameworks include these capabilities if designers and developers choose to use them. Retrofitting an application to have these capabilities, however, can be incredibly difficult—some estimates suggest that retrofitting can consume as much as 40% of the time it took to develop and test the application originally (Source: 2005 study of Web application development times by the World Wide Web Consortium; see http://www.w3.org/International/questions/qa-i18n for an introduction to the topic).

## The Cost of Quality

If you're able to define the costs of poor quality, what will having great quality cost you? Paying for quality typically means paying salaries because improving quality is nearly always a matter of spending more time in key areas. The size of a project, of course, directly impacts the salaries involved: A small project with a single designer and a half-dozen developers will, obviously, have fewer salaries than a major project with 50 developers and a small team of designers. All these professionals are, of course, paid at different levels, and will each spend a varying amount of their time on the project before moving on to something else. The right software development tools can help improve quality while reducing salary requirements.

Thus, to properly calculate the cost of quality, it's perhaps most useful to look at the areas where quality is typically lost. You can then see where in the development process more time is needed, and begin to assess the cost of quality more accurately.

## Where Quality Is Lost

Sadly, quality is rarely lost through deliberate mis-action—that is, nobody's ever rooting for poor quality. Instead, the *opportunity* for quality is most often lost through *inaction*; simple neglect, a lack of awareness, or an honest lack of experience.

### In Requirements

As we've explored in the previous two chapters, the requirements phase of an application is where the biggest opportunities for quality are generally overlooked. The requirements phase is the business' *one and only chance* to specify exactly what the business needs the application to achieve; nearly every major quality fault can be traced back to a set of requirements that simply didn't specify that factor of quality.

#### Ambiguous Requirements

Are your requirements specific? Or, to be more precise, are they specific *from a business level*? I find that legislation is a remarkably good analogy for ambiguous requirements. One state, for example, might pass a law that states "…minors under the age of twenty-one shall not be permitted to sit at a bar where alcoholic beverages are served." It's understood that the idea is to keep children from drinking, but the law says nothing about children who might be seated elsewhere, and that ambiguous wording might lead to someone breaking the spirit of the law being excused because they hadn't violated the letter of the law. The law should have simply stated the *requirement:* "Minors aren't allowed to drink." Let the implementation of that requirement be driven by a clearly stated *intent.*

This is true in software requirements, too. Don't state that "UIs must be efficient" because "efficient" is a relative term that can be measured differently by different people. You might feel that an "efficient" UI requires less than 5 minutes to complete; a developer might feel than 6 minutes is "efficient," and the difference might be millions of dollars in productivity. If the business is looking to realize a benefit, state the desired benefit directly: "The UI must permit a user who has 6 months' sales experience to complete the input form in 5 minutes or less."

#### Non-Functional Requirements

Non-functional requirements—things that don't directly contribute to the application's function, but are nonetheless important from a quality perspective—are frequently overlooked in requirements. What are the application's desired performance metrics? What languages will it need to be used in? Must it be accessible to employees who can't use a mouse, or who are color blind? None of these can be assumed; anything that is desired by the business must be clearly stated.

Performance is often something difficult to state, but it's so important to create a performance goal that, honestly, it should be a primary consideration when creating the requirements. How long should various operations take to complete from beginning to end? Once a user hits "OK" on a dialog box, how long is the application allowed to take before displaying a response? These performance goals *must* be reasonable: Nothing is instantaneous, and wishing for it won't make it happen. And performance goals should clearly state the environment for which they're written. For example, your requirements document might simply state that:

> Performance goals are stated with the assumption that 1000 agents are using the system simultaneously from both of our offices, and that each agent is using a computer running Microsoft Windows Vista, having at least 2GB of memory and a 1.5GHz or faster single-core processor, and connected via a 100Mbps Ethernet adapter to our existing network infrastructure.

This requirement is unambiguous, and it helps ensure that any testing down the line will reflect this real-world scenario—rather than testing being done on the developers' brand-new state-of-the-art computers on a separate, high-speed network with no other users. You also know that you can expect performance to degrade once more than 1000 agents are using the system, and once your numbers start to reach that level, you'll be able to decide what to do about it.

Understand that performance definitely costs. For example, rather than relying on native database drivers, a designer might decide that, in order to meet performance goals, the application will need higher-performance drivers from a third party. The designer might even want to evaluate several options for drivers, and perhaps even build a small pilot application to directly test drivers' performance. All this costs, so you need to understand the potential savings so that you can balance those costs.

### Details of How a System Interacts

It's easy to state high-level business goals—"This application will accept sales orders"—and to leave unstated the details of the underlying interactions:

> This application will accept sales orders via manual user input. Orders will be entered into a local database, and will be sent in hourly batches to external order-processing partners in pre-defined XML formats. Accepted orders will be placed into an existing third-party data archive and into a new data warehouse database for reporting purposes. Orders will also enter the system from order-entry partners using a variety of file formats (including CSV, XML and others yet to be defined); these orders must enter the system and be treated identically to manually input orders.

This level of detail provides far more information about what the application must be able to do. Designers now know that they need to allow for future data formats, for example, and that the user-input application isn't the only means by which orders enter the system's flow. This is crucial information because it drives a number of high-level design decisions that will ultimately have enormous impact on the final application's quality.

### User Error Conditions

What should the application do when something goes wrong? It's very, very rare for application requirements to detail this—but they should:

- Should errors be logged?
- Should errors be displayed to the user?
- What level of detail will help developers and Help desk agents troubleshoot and ultimately solve the error?
- Is the error clear enough that users can resolve it on their own?
- Should the error direct the user to self-service assistance, such as an online knowledge base?

Further, what potential errors will users make that can be anticipated? Can those errors be resolved internally?

A good example of this is the spelling-suggestion used by the Google search engine. Search for "softwar security" and Google will perform the search, but ask, "Did you mean: *software security*?" The designers anticipated that users might make spelling errors, and defined an error condition within the software to suggest alternative searches based on suggestions from a spelling dictionary. This highlights the error to the user, allows them to continue working on their own, and provides a path to resolve the error without involving support resources.

## In Design

Design is where the business' requirements are translated into technical directions that developers and testers can follow to produce the final application. You should be crystal clear on one thing: *The design will not achieve any level of quality that is not specified in the requirements.* That said, it's possible for designers to overlook elements that contribute to poor quality, and the next few sections will highlight some of them.

### Network Latencies and Bandwidth Constraints

Designers can easily fall into the "perfect world" view, where they forget about real-world constraints such as available network bandwidth and latencies. In many cases, the designers feel they have little control over these elements, and that they're justified in ignoring them. In fact, the designer should review existing network conditions and specify a design that guides developers into working within those conditions. For example, the designer might specify a maximum amount of data that can be retrieved in a single call to a database, or might specify that developers plan the application to recover from repeated network timeouts due to high latency. Designers can be driven to consider these constraints by specifying the constraints and real-world conditions in the requirements.

### Scalability

Designers can lose track, or often be unaware, of how the application is expected to grow. An application that works well for one user—say, an application founded on a file-based database such as Microsoft Access—might stop working completely when faced with a few hundred users. Designers need to consider not only the current user workload but also the *projected* user workload for the future—and they should get those projections directly from the requirements. Using the requirements, the designer should be able to specify technologies and techniques for both programming *and testing* to ensure the desired level of scalability. The test design is especially crucial. You can't always take performance measurements from 10 users and multiply to predict performance for 1000 users; computer performance doesn't work like that, and the designer needs to create tests that will allow the application's performance to be accurately tested for the expected load.

Understand the potential cost of that, though. It's not usually possible to accurately test an application that is intended to have 10,000 simultaneous users, for example, without having thousands of testing machines—hardly practical. In order to guarantee that level of scalability, a designer might take an entirely different direction with the application's entire architecture, creating an application designed for a thousand users—which *can* be practically load-tested—and then running multiple instances of that application in parallel, potentially on segregated networks. That type of design is much more scalable, and it's possible to ensure a level of scalability; however, that type of design might be much more expensive than less-easily load-tested designs. Be sure you're specifying—and designing for—realistic, practical levels of scalability.

### Heterogeneous Platform Requirements

Does your design specify that the application be able to run across heterogeneous platforms? Will it need to communicate with both Microsoft SQL Server and Oracle databases? Will it need to run on Windows and Mac computers? Will it accept data from the Internet, where nearly any type of computer might be sending the data?

In the age of the Internet where all computers can communicate with ease, we often take it for granted that all computers *can* communicate with ease; such isn't always the case. The Internet *specified* cross-platform capabilities from the outset; if your requirements don't do so, designers might select technologies and techniques that aren't as cross-platform as you ultimately require.

But be careful: Platform-specific techniques and technologies are often easier and cheaper to work with. If your application will *only* need to communicate between machines running Windows, then designers are right to select for Windows-specific technologies because doing so makes developers' and testers' jobs easier—and therefore less expensive. Forcing cross-platform operation may force a decision to use something more complicated, and therefore more expensive.

### Security Constraints

Designers will not always consider security beyond the very basics, if even that. Your requirements need to specify the operating conditions and security requirements in some detail so that designers can specify appropriate measures. Will the application be accessible to the public? If so, the designer might need to take significant steps to protect against attacks originating on public networks—and those steps might add significant time and cost to the application. Is the application's data subject to specific security requirements? If so, the designer may need to adopt a more complex—and expensive-to-develop-and-test—security model to meet those requirements.

With a good set of requirements, the designer should be able to be very specific about what the developers should do. The design should include a synopsis of the business requirements, and then go into a great level of detail about how the application's security will work. This design must include security interactions across all components of the application, including back-end and middle-tier components, and ideally each portion of the design should explain how that portion contributes to the original business requirement.

## In Development

With a good design, it's tough for developers to go wrong, but not impossible. A key is to ensure that developers have access to the original business requirements—which should always be the arbiters of any conflict or compromise—and that the design specify the ways in which the particular design meets the business requirements.

With a proper design, and with developers who follow that design, nearly all the following can be avoided. However, these are particular areas to pay attention to because they're details that are either often left to developers or areas where developers are most likely to diverge from the design intentions and statements.

### Code that Assumes a Particular Set of User Interactions that Do Not Meet End User Needs or Expectations

Developers rarely come from the side of the business where their applications will actually be used; as a result, applications frequently work the way the developer thinks it should, and not the way the user thinks it should. Take a simple order-entry application: The developer thinks of this as an order, as a customer, and as a set of items within the order. This is a data-centric view of the universe, and it might lead to an application where the user selects a customer, enters order details, and then selects items to be added to the order. Sensible enough, perhaps, but maybe not realistic. The sales user might instead want to start by capturing the items to be ordered so that they can engage in add-on sales and upselling, and by getting the information that is at the top of the customer's mind—what the customer wants to buy. Selecting the customer and entering order information might be last on their list as necessary chores to be completed, but that might bore the customer and reduce the sales opportunity.

A good design will provide detailed workflow diagrams and possibly even UI mock-ups to guide developers; it's rare, however, to find a design that doesn't leave the developer with some assumptions to make—and developers will assume that everyone using the application will think like them.

### Code that Does Not Handle Error Conditions Well

Because designs rarely specify how to deal with errors, developers are often left to their own devices. Ambiguous error messages such as, "Error 52" are typically the best you can hope for; more commonly, developers won't try to anticipate errors and applications will crash, create inconsistent data entries, or do other undesirable things. This is an area where experienced development managers and senior developers can really earn their pay: reviewing code and spotting potential error conditions and ensuring that they're handled properly—according to the design, or at least according to best practices. Automated tools can also help spot certain types of error conditions and help developers ensure that those conditions are handled gracefully within the code.

### Code that Does Not Consider Boundary Conditions

A *boundary condition* is an unexpected condition resulting from data exceeding some predefined limit. If a user enters a 10-digit invoice number, for example, but the system only supports up to 8 digits, what will happen? A *buffer overflow* is perhaps the most dramatic example of a boundary condition: An application sets aside 8 characters in memory for the invoice. Immediately following that are 8 characters for the order total. A user enters 10 digits for the invoice, and the program—without performing boundary checks—stores those 10 digits to memory, passing the 8-character *boundary* and writing the last two invoice digits into the first two characters of the order total—potentially wreaking havoc.

This is one area where best practices, code-quality tools, and careful testing saves the day. A design might specify that the application reject invoice numbers longer than 8 characters—that is, after all, a potential user error condition; but it's often difficult for designers to be that specific with every piece of data entered. Instead, static code scanners can often detect at least the potential for this type of problem, and best practices tell developers that *all* input should be boundary-checked before storing it; managers and testers can help in ensuring those best practices are followed.

### Code that Consumes Resources and Results in Scalability Problems

Developers are often fortunate enough to work on fast computers and have access to the latest technologies; unfortunately, that means they're often not considerate of the computing resources where their applications will actually run. Although it's up to the designer to specify technologies that can scale to the level specified in the requirements, it's up to developers to properly implement those technologies in ways that minimize resource consumption. Once again, best practices, smart management, and development tools can be invaluable in helping to do so.

### Code that Has Security Flaws

Although boundary checks can help prevent certain types of security flaws, other kinds of security problems still exist, many of which rely more specifically on logical flaws in the code. For example, an application's internal security mechanism might assign users a numeric access level; the code might also assume that anyone with a level of 9 or greater has access to anything. Later in the development cycle, access levels 10, 11, and 12 are added—but the original assumption of 9 being the "super user" isn't addressed in the code, resulting in a security flaw. The programming constructs that result in this type of flaw are common, and this type of mistake is an easy one to make. Thorough code review and even more thorough testing are your best bets for catching this type of problem, as are best practices that advocate more restrictive approaches to security.

In addition, the designer can help developers avoid these errors by more clearly specifying the security architecture, and by—when possible—relying on security mechanisms inherited from other technologies. For example, rather than building their own security model into a data-access application, designers might rely on the more robust security already offered by the back-end database system, and simply specify that the application be prepared to deal with "access denied" and other related errors that might be returned by the back-end.

### Code that Does Not Conform to Standards for Internationalization

We've already discussed how retrofitting an application for localization can be far more expensive than building localization in to begin with; if you make the decision to build a *internationalized* application (that is, one that can be localized into different languages), you also need to make sure developers follow the necessary practices *throughout* the application—icons, error messages, and nearly anything else that a user might lay eyes on needs to be programmed in a way that permits later localization. It's easy for developers to stick in "temporary" error messages as they're developing and unit testing, and to later forget to pull those messages and replace them with the proper, internationalized code. Code review, testing, and programming tools can help make it easier for developers to do the right thing and to catch instances where they don't.

### Code that Does Not Meet Expectations for the Information to be Delivered

Screens that don't show enough information or reports that don't contain the right information are often perceived as coding problems; however, in reality, they can frequently be traced to poorly defined requirements. If your application's requirements *are* well-defined, then testing for those requirements will reveal instances where developers didn't adhere to the design or where the design didn't live up to the specified requirements.

## In Testing

Most organizations feel that testing is where quality begins. In fact, it's certainly where you can see poor quality most visibly arrested, but it's really just the *last opportunity* for quality out of a long software development life cycle. It's an important last opportunity, though, and many organizations don't maximize that opportunity.

Testing is often performed by a QA department, and there's a reason it's called Quality *Assurance* and not something like "Quality Implementation." On its best day, QA—that is, testing, specifically—can only *catch quality flaws.* It can't actually *improve* quality; flaws must be turned back over to developers, who repair the flaw. Testing cannot *add* quality, but poor testing practices *can* detract from quality by failing to catch flaws.

### Not Testing Business-Critical Processes

If the business requirements are what drive the application's design and development, it should drive the testing, too. Regard the requirements document as a checklist against which the entire application should be tested. Test the application as a *real user* would use it, using realistic data (both good and bad), under realistic conditions (even if simulated), and for a realistic number of cycles (don't enter one order when a real user would enter 50 orders per day). Unit testing and other levels of testing are great and necessary, but the ultimate measurement of quality is testing the entire business processes that the application was intended to implement.

### Not Testing for Usability

Is the application easy to use? Does it meet its usability requirements? Again, the requirements drive the vehicle, and without good requirements, it's very difficult to say whether the application is "usable."

Make sure usability testing focuses on real-world usability, too, not just theoretical usability. Sure, users might agree with the "theoretical" workflow for 80% of the time, but the application needs to remain usable for the other 20% of the situations, too. Observe users, assemble realistic test data and usage scenarios, and make sure—again—that everything matches back to what the original requirements specified.

### Not Testing for Performance and Scalability

Are you tired of hearing that the application's requirements should specify performance and scalability goals? If so, good—you're getting the message. As with everything else in the requirements, testing is the last chance to verify that the requirements have been met, and nowhere is this more difficult than in dealing with performance and scalability.

Automated testing tools can help "load test" applications, but these load tests are *very rarely* identical to actual production application load; production loads are random and "peaky," meaning they're almost never as evenly distributed as the load a load-testing tool might create. For example, a load-testing tool might reveal that your back-end database performs well with up to 5000 users; under production conditions, however, you might see poor performance when the load *suddenly* peaks to 5000 from a much lower level, rather than being evenly applied all along.

The goal is to consider all the factors of a production environment and try to simulate them *as closely as possible* in testing. You should also "stress test" by applying unreasonable (and unrealistic) loads to the application as a whole, and to its various components, so that you have upper-limit metrics on performance. All these different types of performance testing can help spot flaws in the application; all the testing metrics should also be retained to help ongoing production health and performance monitoring of the application.

### Not Testing Error Conditions

Error testing is often regarded as testers "trying to break it," and testers often get a certain amount of glee in reporting bugs back to developers. That's fine, and it's a necessary part of testing, but error testing isn't just "trying to break it." Error testing should *also* include methodical attempts to feed the application bad data, take the application down *every* possible combination of workflow paths, and so forth—testing to make sure that what *will* become common occurrences in the real world are tested, caught, and fixed during the testing phase.

### Not Testing Broadly

It's easy—especially with automated testing tools—to get caught up in testing just a few specific things, especially if those things are being problematic and revealing a lot of bugs. Testers live on bugs, and if they're producing a lot of them, they feel they're really doing their jobs—and they are, just not their *whole* jobs. The *entire* application needs to be *repeatedly* tested. You test module A and find no problems, but find 10 problems in module B. Are you done testing module A? No, because the fixes in module B may affect module A; you need to focus on the *entire application.* That may be boring, which is why those automated testing tools exist, but it has to be done to ensure quality.

## Barriers to Quality

So why don't we all just get serious about quality? What's stopping organizations from having amazing levels of quality in every application? There are two distinct areas: barriers within the IT organization itself, and barriers that come from the broader business. It's truly important to recognize that these barriers exist to some degree in nearly *every* organization in the world, or that these reasons at least have the *potential* to exist within any organization. These are the things you *will* have to fight to achieve quality, and you will continue to fight them until the organization formally recognizes the impact of these things on quality, and the benefits in removing these barriers and allowing quality to become a part of the business' daily life.

## IT Barriers

The IT organization faces barriers that fall into two broad categories: resources and knowledge. Both contribute to quality, although the resources category is somewhat easier to see and fight—and it's why many IT organizations feel that resources (or lack thereof) in some form are the *total* quality picture.

### Tight Budgets

In business, resources equal money; tight budgets mean tight resources. This might take the form of "not enough developers," "not enough time to design the application properly," "not enough testers," and so forth. In some cases, tools can be cheaper than human resources, and tools can in some instances make up for a small lack of human resources. Automated testing tools, for example, can help reduce the number of testing staff needed while still helping to improve application quality. But automation can't replace a skilled application designer taking the time necessary to produce a quality design, so automation is *not* the end-all, be-all of better quality.

The IT organization needs to have a frank and open dialogue with management about the real costs of building a quality application, and a solid understanding of the costs of *not* building a quality application. To overcome the budget barrier, business management *must* be able to commit to a given level of quality, be able to communicate that level of quality, and be able to measure the quality actually being delivered.

### Don't Have Understanding of Quality Best Practices

Inexperienced development teams often lack a good understanding of quality best practices. It takes time, for example, to code in boundary checks for every input—and frankly, it's boring coding. Why do it? Sure, everyone knows it's a "best practice," but what's the cost in not following them? New developers may even lack fundamental knowledge of what best practices are. In all cases, the key to removing this barrier is education: classes for beginning developers, reading materials, and automated coding tools that can help developers spot areas where best practices belong, explain what those best practices are, and help implement those best practices in code.

### Lack of Business Involvement or Accountability

It's a little stunning, sometimes, to see how little the business involves itself with application development that is intended to benefit the business. Let's go back to the home-building analogy. If you, the homeowner, aren't in on blueprint design, and if you're not showing up at the job site periodically, why in the world would you think to complain when your house wasn't suitable for your family?

Yet businesses often have a turnkey attitude, handing off projects to IT with a demand only that they be accomplished quickly and cheaply, and investing little management time in ensuring that IT can actually do so. Typically, poorly defined requirements are the glaring neon sign of an uninvolved business who doesn't care what IT does so long as they somehow get it right—without business leadership.

This barrier is easily solved in theory: IT simply doesn't begin projects that don't have clearly documented business requirements that communicate the end benefits the business wants to realize. Smart CIOs and CTOs understand that they'll be held accountable for what IT does, and they make sure that the business is driving what IT does. Smart development managers won't begin a project until the business is willing to invest the time to create a detailed set of business-level requirements.

Let's be crystal clear—This is the single most important and challenging barrier that IT faces. This is what makes or breaks quality. Every other IT-related challenge can be dealt with internally—whether it involves juggling budgets, educating developers, or what-have-you; unless the business is involved enough to produce requirements, the project will ultimately produce poor-quality software. Period.

### Lack of Business Process Knowledge

IT knows IT; it may think that it understands what the business does and how the business operates, but that's rarely the case. There are nuances and details to every business project that are never captured in flowcharts and requirements documents. IT must recognize their lack of business process knowledge and must instead seek that knowledge from the people who will be using the application—not their four-times-removed manager but the actual people who will be touching the buttons. Nuances are a key factor in quality, and those nuances come only from on-the-job experience. By openly acknowledging their lack of business process knowledge at a nuance level, IT can open conversations with business process practitioners to ensure a quality application.

### Collaboration Is Difficult Between Practitioner Silos

Another problem IT has is its own internal silos: developers like things one way, database administrators like them another, and systems administrators prefer things a third way. Typically, this inter-disciplinary tension is a good thing because it helps bring many different viewpoints to the table; a good application designer and project manager can and should be the arbiter between disciplines, favoring nothing except the quality of the final application. The project manager in particular should always be thinking, "how does this help further meet the original business requirements?" and resolving any disputes or disagreements in favor of those business requirements.

## Business Barriers

Businesses place plenty of their own barriers in front of quality, and recognizing these are the first step toward eliminating them. Let's look at a few of the common barriers, and discuss where they come from and how they can be avoided.

### Lack of Time to Spend Working with IT on New Application Initiatives

Requirements, requirements, requirements. Has it been said enough? Be completely clear about one thing: *IT cannot produce a requirements document by itself.* It *must* come from the business. The business is paying for this application, after all, and the business presumably has some reason or reasons for doing so. *Write those down.* Write them down in a good level of detail—and those are your requirements. But if the business isn't willing to spend the time working with IT to develop those requirements, the business would be best served by *not undertaking the development project at all.*

Let's forget new home construction for a moment and talk about kids. You send your child to school, and they bring home a straight-D report card. Did you ever make it clear that "D" wasn't acceptable? No? Then don't yell at the kid—they had no expectations, no *requirements,* and so they did what they wanted to.

> **Bob: No Clear Requirement**
>
> One of the most public failures in the software industry was the mid-1990s release of Microsoft "Bob," an application that aimed to make Windows easier to use for non-computer experts by offering a more familiar, real-world set of analogies to computing tasks. "Bob" failed miserably, but not because it had a lot of bugs, or was hard to use, or was too expensive, or anything we might typically associate with "quality." Instead, "Bob" simply didn't meet the needs of the audience. The people who were expected to pay for "Bob" didn't care about the things "Bob" did—it solved a problem that, for them, wasn't really a problem. "Bob" is an excellent example of an IT department—Microsoft, in this case—embarking on a problem that they had been told about, which is that computers were too hard to use. But they didn't get a clear set of requirements from the people who told them that, and so the final result—a computer company's vision of how to make computers easier to use—didn't even remotely match the actual needs.

Business *must invest the time* to work with IT on new applications. There's no option, and any sane IT department will politely decline to proceed until the business *has* the time.

### Lack of Money to Invest in Better Quality Practices

Quality, as we've said more than once in this chapter, costs. If the business can't afford to pay the price, the business can't have the quality. Period. That said, most businesses feel they can't afford quality because in many respects, they can't understand it, measure it, or even measure what it costs *not* to have the quality; those are areas where IT can and must help. By showing the business how to measure quality and how to attribute costs to a lack of quality and by helping the business gain insight into quality throughout the IT organization, the business may realize it *does* have the funds for quality, after all.

### Expect Quality to Be There

Business managers may not even think of quality, or if they do, they assume it comes from having highly paid developers. As we've seen, that's simply not true. It may therefore be incumbent on the IT organization to demonstrate, numerically, where quality has to come from, and what the costs of poor quality are. Point to past projects that had quality issues, and explain in detail how various poor-quality elements came from a lack of business involvement, education, or other means. Above all, help management understand that quality doesn't "just happen," and that it's a measurable, definable set of practices that can be implemented and followed to produce quality—but that quality is as much a "product" as the software application itself.

## Benefits of Quality

Obviously, quality is good. Nobody reasonable will ever disagree with that statement. But *why* is quality good? Because if there's no *reason* for it to be good, there's no reason to pursue it and produce it. Knowing what to expect at the end of a high-quality project is what drives you to make it a high-quality project in the first place; just as we've looked at the barriers both IT and businesses place in front of quality, let's now look at the payoff both IT and the business can expect to achieve by removing those barriers.

### IT Benefits

When quality is achieved, IT becomes more business-driven, more consistent, and more proactive. Ultimately, these all benefit the business, but they have an immediate and more noticeable impact at the IT organization level.

### Lower Help Desk Costs

This is perhaps the easiest metric for software quality: Have the Help desk calls gone up or down? Good software quality means users aren't complaining about bugs, performance, missing information, or anything else—meaning the Help desk spends less time dealing with the software, which means IT costs immediately go down.

### Free Up the Best Resources for New Frontier Projects

Poor quality software typically consumes the time of some of your best and brightest people—the ones who can deal with troubleshooting a bad application in midstream, who can fix it quickly, and who can test and deploy it reliably and rapidly. Wouldn't you rather have your best and brightest working on new, important projects that will have real impact on the business? It's almost certain that *they'd* rather be working on new projects than troubleshooting and repairing old ones; it's difficult to imagine the business *not* realizing a benefit in having your best people working on new projects to further the business' capabilities. Good software means good people can move on to new projects, continuing to improve the business rather than fighting fires and trying to keep the business afloat.

### Eliminate Unnecessary Infrastructure Investments

Poor quality software is a direct driver of unnecessary upgrades to networks, servers, databases, disaster recovery systems, and more. An application with poor network performance is practically impossible to re-architect and improve; it's usually more practical to simply upgrade the network to deal with the application's actual real-world requirements. But it would have been cheaper by far to build a good quality application from the outset—an application that *used* the *existing* infrastructure rather than *abusing* it.

## Business Benefits

When quality is achieved, the business saves money. It really is that simple: Quality is something that costs, but the costs are typically traceable directly to end benefits. Or at least the opposite is true: Good quality software keeps business from being bad.

### Higher Revenues and Market Share

Most businesses can point to the lost revenue and, as a result, the lost market share due to poor-quality software. Software that loses orders, takes to long to implement good customer service, makes employees work harder instead of smarter, and so forth contributes to lower revenues—and it's all due to poor quality. It stands to reason, then, that good quality software—applications that retain data, make data entry easier, connect with customers more meaningfully, and so on—will help raise revenues.

### Increased Customer Loyalty

Sometimes, businesses get the idea that they're fighting to get money from customers. In fact, customers typically *like* doing business with good organizations—ideally, doing so makes customers' lives easier in exchange for some money. What customers *don't* like, and why businesses often feel like they're fighting customers, is businesses that make it tough to do business.

If you call to book a flight with an airline, and it's a hideous process—they take forever to find the right flight, have to keep asking you for your credit card information, and take hours to email a confirmation—are you likely to keep them at the top of your list for future travel? No, they've lost your loyalty, and in many cases, the reasons can be traced to poor-quality software. Software is intended to make it easier for businesses to do business; poor-quality software makes doing business difficult, and makes retaining customers painful.

### Improved Regulatory and Industry Compliance

Quality software has fewer defects that lead to information leaks and security breaches, meaning businesses have an easier time meeting government- and industry-mandated rules. Quality applications provide reporting and auditing tools as required, and the business can treat compliance as a *part* of the business, not something that occurs *on top of* the business.

### Lower Operating Costs

When quality becomes a science within the organization, operating costs simply go down. Software has fewer defects and aligns with business needs, which nearly always revolve around lowering costs by increasing productivity. Lower operating costs means better margins, which means better profits, which means everyone's happy because the investment in quality paid off.

## Summary: Approaches to Quality

So how do organizations approach quality? There are really four levels, which I categorize as:

- Quality as a Hobby—Organizations without a quality team, who rely primarily on user acceptance as their quality metric. They tend to view automation (such as automated testing) as the key to better quality.
- Quality as an Effort—The organizations that have an established quality team, but one with little experience. Usually there are some QA tools in use and defect tracking is in use, but there is difficulty in establishing quality goals.
- Quality as a Profession—A mature QA team, possibly more than one, exists in the organization. Different teams use different tools and techniques, and releases across teams are of inconsistent quality. Executives have little insight into quality, and quality is not typically aligned to business objectives.
- Quality as a Science—The ultimate level, where consistent quality across projects is apparent, and where management has clear insight into quality levels, problems, and processes.

In the next chapter, we'll see where your organization fits into these levels, look at what comprises each one, and lay out some practical, achievable action items that can lead you from one level to the next.