Realtime publishers

The Definitive Guide™ To

Quality Application Delivery

sponsored by

MICRO FOCUS®
Leading the Evolution™

Don Jones

Introduction to Realtime Publishers

by Don Jones, Series Editor

For several years now, Realtime has produced dozens and dozens of high-quality books that just happen to be delivered in electronic format—at no cost to you, the reader. We've made this unique publishing model work through the generous support and cooperation of our sponsors, who agree to bear each book's production expenses for the benefit of our readers.

Although we've always offered our publications to you for free, don't think for a moment that quality is anything less than our top priority. My job is to make sure that our books are as good as—and in most cases better than—any printed book that would cost you $40 or more. Our electronic publishing model offers several advantages over printed books: You receive chapters literally as fast as our authors produce them (hence the "realtime" aspect of our model), and we can update chapters to reflect the latest changes in technology.

I want to point out that our books are by no means paid advertisements or white papers. We're an independent publishing company, and an important aspect of my job is to make sure that our authors are free to voice their expertise and opinions without reservation or restriction. We maintain complete editorial control of our publications, and I'm proud that we've produced so many quality books over the past years.

I want to extend an invitation to visit us at http://nexus.realtimepublishers.com, especially if you've received this publication from a friend or colleague. We have a wide variety of additional books on a range of topics, and you're sure to find something that's of interest to you—and it won't cost you a thing. We hope you'll continue to come to Realtime for your educational needs far into the future.

Until then, enjoy.

Don Jones

Copyright Statement

Chapter 1: Introduction

Application development executives told Forrester that their biggest challenge for delivering new and updated in-house applications is quality (Source: *Application Development Executives Need Better Visibility to Become More Reliable Business Partners*, Forrester Consulting, May 2007). In the same study, 43% of those executives said that they planned initiatives to reduce application development cost. 35% planned to focus on quality improvements, and 28% planned to work on their time-to-market metrics. When it came to quality, they said their biggest problems, aside from the perennial problem of finding enough staff, were time pressures, requirements, complexity, and processes. Poor requirements have been cited as a root problem in software quality (Source: *The Root of the Problem: Poor Requirements,* Carey Schwaber et. al., September 2006) since time immemorial, most commonly in the infamous "scope creep" that negatively impacts so many software development projects.

You've seen it. We've all seen it. Delivering high-quality applications is incredibly difficult. Not only are applications commonly perceived by users as buggy, but they're often perceived as simply working incorrectly—meaning they don't meet their requirements. Unfortunately, there's no simple, easy button that you can push to improve software quality—despite some manager's uninformed perception that an add-on suite of software testing tools will solve the problem overnight. Instead, you'll find that the ultimate answer is a complete overhaul to the way you look at software quality, and a more mature approach to the way you develop software applications, starting at the beginning of the development project. That's what this guide is all about—examining where quality problems begin to enter a project, and defining ways to improve quality throughout the life cycle of the project.

The goal of this chapter is to provide a sort of executive overview for the entire guide. In this chapter, I'll outline the problem and the basic steps toward a solution; subsequent chapters will dive into much greater detail on these steps and help you build a clear picture of what you need to do in order to bring quality applications to your organization.

What Is Quality, and Who Cares?

Few in the IT industry could argue that there's at least a perceived problem with software quality. Few, however, like to acknowledge how difficult quality can be to attain. For example, software such as Microsoft Windows is often perceived by users as frustrating, full of bugs and instabilities, and not working the way people want. Windows Vista has made headlines with the number of users who perceive it as slow, unstable, and not doing what they want. For years, a maxim with Microsoft applications has been "wait until the first service pack" to implement. Now consider: If that's the level of quality perception that the world's largest software company can produce, what hope do you have for doing better?

Perception, of course, is a key word. Application users seem to love to hate the applications they use. But there's a reason for that. Actually, there are several reasons:

- **Bugs**. The easiest things to point at in terms of poor quality perception are bugs, software defects that prevent the application from working in the way in which it was designed. Bugs creep into products for a number of reasons: Poor requirements, poor coding practices, inexperienced programmers, poor software designs, badly documented underlying APIs, and pure human error.
- **Performance**. How many times have you called a company and waited on the phone while an agent old you, "sorry, my computer is slow today?" Slow-running applications are a leading cause behind the perception of poor quality in applications, yet performance is rarely a design consideration early in application development projects.
- **Function**. Have you ever watched a waiter at a restaurant struggle to enter an order into their point-of-sale terminal? Or waited in line at a cash register while a manager was summoned to deal with a routine problem, like voiding an item from an order? "Did the people who wrote this ever actually work in this environment?" is another common refrain from application users, driving back to the root problem of poor requirements in software development projects.

Traditional Quality Indicators

One barrier to improving software quality is that we often don't do a very good job of measuring quality in the first place. Many organizations simply *don't* measure quality at all, instead relying on anecdotal evidence—that is, stories from users, often voiced over the phone to Help desk technicians or managers when the user is frustrated by an immediate problem they're experiencing. Other organizations tend to measure quality in a very one-sided manner, either by simply counting the number of bugs encountered by users who are using the product or by measuring the number of Help desk calls the product causes. Neither of these "metrics" actually bears any real relationship to the actual level of quality in the product:

- Bugs reported by users are sometimes not bugs at all but rather deliberate outcomes of the software's design. Although these outcomes may ultimately be undesirable, tracking them as code defects is unfair to the software's developers and testers, and inaccurate in terms of helping to fix the problem.
- Most bug-tracking systems are full of unintentional duplicates, making many software applications seem worse than they actually are. Problems are described in different ways by different users and entered into the system by different personnel. This duplication makes it harder to fix the problems because the details of the problem end up being scattered across multiple bug reports. Also, a problem may exhibit several undesired behaviors and symptoms, giving the impression of multiple problems, when really they stem from only one cause.
- Anecdotal evidence nearly always reflects the negative of a product: You don't get many users calling in to praise a product. As a result, organizations can easily develop a lopsided view of their applications' quality. Further, every organization tends to have a few "squeaky wheels" who do most of the anecdotal reporting, meaning that perception of the software becomes even more lopsided and biased toward the views and needs of a few particular users.

Perhaps the worst problem with these metrics is the myriad of issues that they don't measure or address. Performance is rarely considered except from anecdotal reports of "this application is running slow." So what should you be measuring when you consider the quality of your applications?

- **Bugs.** Certainly, software defects are a concern and should be tracked. Some organizations define an acceptable number of defects for an application, which is fair when an application is being cranked out with insufficient staff and under heavy time pressures. However, I'm reminded of an interesting story told by old IBM engineers during the time when IBM was just beginning to outsource microchip production to companies based in Japan:

  IBM ordered 100,000 chips on the first order, and specified a defect rate of 5%. It was a simple miscommunication between two different cultures: IBM understood that a certain number of chips would simply not make it through the automated production process correctly. Their Japanese supplier, though, hadn't yet been introduced to the concept of "acceptable failure." When the first shipment arrived at IBM's facility, it contained a note: "Enclosed are the 100,000 microchips. Also enclosed are the specified 5,000 defective microchips, packaged separately."

  This story has been told and re-told so many times that its origins are lost, and indeed it may be entirely apocryphal, but it tells an important lesson: The goal should always be for 100% quality and 0% defects. Anything else may be understood as inevitable, but processes should be created to catch those defects before they ship in the final product.

- **Performance.** Software development executives, managers, testers, developers, and users are tired of hearing "this application is slow." Measuring performance as an aspect of quality, then, is clearly an excellent idea. But performance typically comes across as raw data, not as processed information. Knowing, for example, that an application can process 100 transactions an hour isn't useful unless you also know that the application's user, left unhindered, could process 1,000 transactions an hour—meaning 100 is pretty poor performance. In other words, an application needs to have performance metrics designed from the outset with a clearly defined performance goal that can be checked and re-checked throughout the development process, and used as a benchmark when users complain that the application isn't running quickly enough.

- **Tasks.** A key element of any application is whether it properly performs the job for which it was created. Unfortunately, this too often is measured from a purely end-of-flow perspective. For example, "this application is designed to place new sales orders into the database" is a good high-level description of the application's purpose, but it's a terrible statement in terms of quality measurement. Such an application could require users to navigate the most byzantine imaginable processes in order to enter a new order, yet the application would be seen as "functional" provided the sales order eventually got into the database. Instead, application quality must encompass specific task flows and business processes so that the application allows users to work more efficiently and more productively.

Clearly, we need to begin measuring different things in order to accurately assess the quality of our software. The most important of these—performance and the ability to efficiently accomplish tasks—are things that have to be designed into the application long before coding ever begins. This drives right back to the oft-stated "root of all problems"—*requirements.* It also drives quality away from a "let's measure defects" attitude and more toward an attitude of quality as a business metric.

Quality from a Business Perspective

It's incredibly easy to lose track of why a particular software application is being created in the first place. Developers get bogged down in the details of their code, testers get focused on creating test patterns, and managers begin to deal with staffing, scheduling, progress reports, and more. Amazingly, few organizations that have produced in-house software can ever produce written evidence on why they produced the application. Some can perhaps produce a vaguely worded "mission statement" or "vision" for the application, but it's rare to find an organization that can succinctly state the specific business needs that the application was meant to fulfill.

You might think that's an easy task: "We need an application that will allow our sales people to enter sales orders." Simple. But is that really the problem? Are salespeople entirely incapable of entering orders today? Probably not—few applications are created to enable entirely new business functionality; the majority of applications are generally intended to *improve business processes.* Which means a better statement of the application's purpose might be, "we need an application that will allow our sales people to enter sales orders more quickly and from a variety of platforms including their desktops, laptops, and mobile devices." This begins to state the actual business needs through implication: "Today, salespeople spend too long entering orders, and perhaps can't do so unless they're at their desks."

Ultimately, businesses pay for software applications to be created. They do so because they expect to see tangible benefits—typically stated as a return on investment (ROI)—from the applications. From a business perspective, a "quality" application is one that provides tangible benefits quickly and consistently. That means an application that is, incidentally, free of defects, running at an acceptable performance level, and efficiently accomplishing the task for which it was designed—perhaps the three key broad areas of quality perception that I discussed earlier. But quality from a business perspective is impossible to achieve unless the business' perspective—that is, the things the business expects the application to achieve, and ideally the problems that the application is meant to solve—is clearly stated up front. That brings us right back to the application's *requirements,* and how poorly written requirements are a root cause of quality problems.

The Software Development Life Cycle

Let's take a moment to get on the same page with some terminology. This guide isn't about teaching you a new software development management framework; you doubtless have one, and you've doubtless worked with many over the years. Instead, let's just define a few common terms to describe the major elements present in any software development management framework so that we have a common basis for communication.

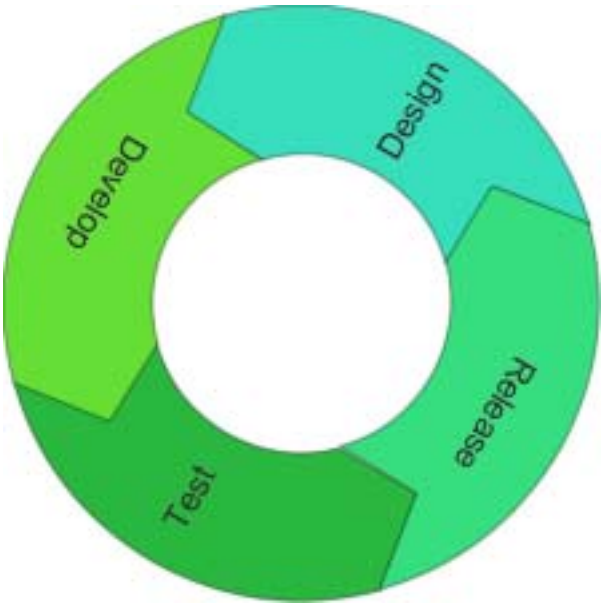

Figure 1.1: Generic software development life cycle.

Broadly speaking, most of us think about software development as occurring in four phases, as Figure 1.1 shows:

- **Design,** which is where we decide what the software will do. Many organizations break down this phase into two important sub-phases:
    - **Requirements,** which is where we gather the facts about what the application is supposed to do. This is one of the most important phases of software development, but it's often given short shrift and accomplished in a bit of a hurry.
    - **Logical and Physical Design,** which is where a software architect defines the various entities, components, data flow, and other elements that will guide software development. Ideally, of course, this design maps directly back to the requirements.
- **Development,** which is of course where the actual code is written. Usually, unit testing—that is, individual developers testing their own code under fairly limited uses—also occurs during this phase.

- **Test,** which is where organizations often conduct more complete functional testing to ensure that all the various application components work well together.
- **Release,** which is where the application is released. Ideally, this results in feedback from the application's users, which drives the design of the next version, completing the loop in the life cycle.

I'll use these general terms throughout this guide, and in Chapter 2, we'll focus more specifically on the issues of quality—where it comes from, what quality should actually mean, who cares about quality, and more. We'll also spend just a bit more time further defining this software development life cycle and talking about some of the often-overlooked tasks that should be going on during each phase.

The Costs of Quality (and the Costs of None)

Businesses often wonder how much it would cost to build extra quality into software applications. In fact, I've had at least one executive ask me exactly that during a consulting engagement: "Sure, quality. What would it cost to have some more?" It conjured in my mind a sort of retro-style television ad: "New software applications! Now fortified with extra Quality™!"

No Quality = Cost

Let's start by looking at what having *no* quality costs, or at least what having *less* quality costs. It's easy enough to point at bugs—defects is the polite term—and talk about how much they cost in terms of re-programming, lost productivity, potential damage to the business through lost or incorrect data, and so forth. But I feel there's a bit too much emphasis on defects as a quality measurement already, so I want to look at some of the other, less obvious costs of quality. First and foremost is the cost required to build a software application, with the hopes of getting a tangible ROI, and ultimately *not* getting that return. Even a small application can cost a business hundreds of thousands of dollars in salaries alone, and if that application doesn't create the tangible return that was hoped for, those dollars are simply lost.

I once worked for a telecommunications company that wanted to develop an in-house application for their network integration division. The application was supposed to implement a number of ISO-certified business processes encompassing network design, customer service, and more. A dozen developers worked on the application for most of the 2 years I was at the company, never producing a single release. The project was ultimately shelved at a cost of more than $2.6 million in personnel costs. The primary reason it was shelved? When it neared time for a release, not a single manager in the company could agree that the application accomplished the business tasks they needed, in the way they needed them performed, and at the speed they required. Defects never became an issue because the application's first release had a quality of "zero" in every way that mattered to the business.

Quality = Cost

So not having quality obviously costs; but having quality costs, too. There's a definite balance that you have to strike between cost and quality. For example, that telecommunications company probably spent about a month gathering requirements—obviously not long enough. But they easily could have spent years doing so, and ultimately wound up with a better piece of software, perhaps 5 years down the line. To turn back to the old standby of software defects, it is always true that more testing will catch more defects. But at a certain point, you start finding fewer and fewer bugs per hour of testing, so there's definitely a point of diminishing return.

The Search for Quality

So where is quality ultimately lost? The key to finding missing quality is remembering that quality isn't simply a measure of software defects or lack thereof. Quality enters a project from the very beginning or not at all:

- **Requirements.** Too often, requirements are ambiguous or vague or don't accurately reflect or describe real business needs. Without a clear, unambiguous, written description of "why we're doing this in the first place and what the system needs to do to accomplish the business goal," you can never achieve the best level of quality. Requirements often don't consider performance metrics, either, or other non-functional needs such as security and business process.
- **Design.** Designs too often don't take quality into account. They're focused on the end-function—that is, putting orders into a database, for example. They don't tend to focus on the actual processes that users will follow, and designs rarely if ever make performance considerations part of the design.
- **Development.** This is usually the first visible sign of quality failures. Without a clear design and solid, detailed requirements, developers often make assumptions about user interactions, and developers may not focus on performance at all. Poor coding practices also lead to quality issues.
- **Testing.** Designed to be the guarantor of quality, testing often leaves plenty of quality on the table. Testing may not focus on the most quality-critical processes, such as those that are crucial to productivity or revenue generation, meaning there's a disconnect between what's been identified as a requirement and what's being tested. Usability, scalability, and performance are often only minor testing concerns because testing tends to focus first and foremost on finding bugs. Poor test data and use scenarios also result in testing ultimately delivering less quality than it could.

The costs of poor quality escalate more quickly in foundation software, such as service-oriented architectures, where poor quality in terms of function and performance can be magnified as the architecture is built on. You're only as strong as your weakest link, and poor quality in the foundation software means individual applications will always suffer from inherent quality issues.

Barriers to Quality, Benefits of Quality

So why don't we simply put more quality into everything we do? There are a number of barriers to quality; things that keep quality at bay. Learning to recognize them and deal with them is crucial to building an organization, and processes, designed to deliver quality:

- **IT Barriers.** Tight budgets, a poor understanding of best practices, and a lack of business involvement are common barriers to quality from the IT side. Silos within IT—between engineers, architects, developers, and administrators—also make quality more difficult to build in.
- **Business Barriers.** Businesses often lack the time to spend working with IT to properly define new applications, resulting in poor requirements. A lack of funding is also a barrier to better quality. Perhaps most insidious is an often naïve assumption that quality will simply "happen," resulting in insufficient time being spent *ensuring* quality.

Yet, if these barriers can be overcome, both IT and the business stand to reap significant rewards:

- **IT Benefits.** When quality is built in, Help desk costs go down. IT gains the ability to innovate as better resources become freed for new projects rather than tied down fighting fires. Unnecessary investments are curtailed, freeing budget.
- **Business Benefits.** With quality software, businesses clearly benefit from better efficiency and productivity, leading to increased revenue and increased customer loyalty. Businesses struggling with regulatory compliance issues can struggle less, and overall operating costs can be lowered—both of which are significant competitive advantages.

> **Cross Reference**
>
> In Chapter 3, I'll dive into greater detail on the costs of quality, the barriers to quality, and the benefits of quality.

What Is Your Quality Level?

Analysts love to measure IT organizations by varying levels of maturity, and software quality is no exception. Fortunately, software quality is an area where measuring your maturity, or "quality level," as I like to call it, can be truly beneficial. No organization can go from low quality to amazing quality overnight; it's a process that you have to follow. You can't skip stages of evolution—you need to experience each one because each one sets you up for the next.

> **Cross Reference**
>
> In Chapter 4, I'll present a brief quiz to help you assess your quality level.

Although everyone in the industry uses different terms, I like to break quality into four main levels:

- **Quality as a Hobby.** At this level, you tend to be focused on identifying defects. Testing is for functionality only, and your testing is a discrete event that takes place in the software development process. You have a testing team (even if it's small), and you may even use some tools to help automate testing and defect tracking. But quality tends to come at the end of the process, when all you can really do is identify a lack of it—you can't go back to the beginning, to the requirements, and start building in quality. Quality doesn't include performance, functionality, or business needs; it's almost entirely focused on defects.
- **Quality as an Effort.** At this level, "quality assurance" or "QA" is a standard term. You recognize that quality must be examined throughout the project life cycle, and you have a dedicated QA team—and you probably call them "QA." You're looking more at business requirements and functional acceptance in addition to defect levels. You're using tools to track defects and possibly to track business requirements as well.
- **Quality as a Profession.** Here, you're managing quality more actively. If you're using the term "total quality management," quality has become a profession for you. You're weighing quality and requirements against risk, cost, and complexity, and you're involving the entire IT team in the quality effort—not just a dedicated QA team. Training and business processes designed to ensure quality are in place, and people are using them.
- **Quality as a Science.** The ultimate level of quality ties quality to the IT management and governance model. Quality initiatives and activities are tied directly to business value. Delivery life cycles are more strategic, and you're measuring quality in terms of alignment with business value. Quality is enterprise-wide, and it involves everyone and every process that you work with.

As I've said, moving through these different levels of quality can be quite difficult because doing so involves the entire organization, not just IT. That's an important lesson that Chapter 4 will really hammer home: By itself, *IT is incapable of achieving a very high level of quality.* You need commitment and involvement from the entire organization. Think of it this way: If you ask someone to build a house for you, and then leave them to their own devices with no further details or instructions, you probably won't be very happy with the result. But if you remain involved throughout the process, involving an architect, an engineer, and a *builder* from the very beginning, then you'll have a house that's easier to build, better suited for your needs, and ultimately of higher quality.

Requirements: Quality from the Beginning

In Chapter 5, we'll start from the very beginning—a very good place, as they say, to start. So often identified as the root cause of poor software, we'll spend time looking at what software application requirements should contain, and how your requirements are your highest pinnacle of quality, meaning you'll never achieve any level of quality that isn't present at the beginning, in your requirements.

Business Needs as Quality Metrics

Your requirements documents need to clearly and unambiguously state the reasons for the application *in business terms.* What exactly does the business want from the application and, most important, why does the business want those things? If there are existing business problems, state them, and state their impact on the business. It's not enough to simply say, "salespeople spend too long entering orders." Instead, clearly state exactly what the problem is, its impact on the business, and the desired outcome:

> *Today, salespeople spend an average of 40 minutes entering a single new order, including setting up a new customer in our systems. With an average of 10 new sales entered per day, salespeople spend 6-2/3 hours per day simply on paperwork—leaving little time for actual sales. This means each salesperson is only selling for about 2 hours per day, meaning each salesperson is only about 22% efficient. If each salesperson could be made at least 50% efficient, we could double sales! To accomplish this, a salesperson making 20 sales per day must spend no more than 3 to 4 hours entering those sales, meaning each order must require no more than 5 minutes to enter in an accurate and complete fashion.*

Of course, being able to state the problem and the desired outcome in such clear, *measurable* terms requires a thorough understanding of the problem, and this is where most business' zeal to improve disappears. Business analysis is a tedious, often thankless task that employees often perceive as an investigation of *them* rather than an investigation of the *process* and its *tools.* But by stating the requirement in this fashion, look at all the things that are driven further down the development process:

- Since the time-to-complete-task goal is so clearly stated, the entire design process will be driven to thoroughly understand and improve the workflow used to complete the task.
- Programmers will make fewer assumptions and ask more questions, as they have a clear goal of implementing a workflow that can be accomplished in a specific amount of time.
- Testers have a clear use case to employ during tests, and a clear way of determining the application's ability to meet the original goal.
- Release-related tasks, such as end-user training and documentation, also have a clear goal to meet, and can focus on training users to use the new software in a way that drives the original goal.

Further, the value to the business is clearly stated: The intent is to allow a doubling of sales, an obvious business benefit. That benefit is calculable: If sales today are at $1 million, then the goal is to enable $2 million in sales. If the cost of doing so is $1 million, then it may be worth it, as the cost is repaid in a single year. If the cost of doing so is $5 million, some compromise may need to be reached, since the cost so significantly outweighs the benefit. Understanding the real value of the application is useful in making decisions and weighing the balance between risk, complexity, quality, and other factors. For example, if the developers have created a process that requires 7 minutes to enter an order, and estimate that it will take 3 more months to refine the process down to 5 minutes, management may well decide that 7 minutes is good enough. It doesn't meet the original goal, but it does meet a goal that's acceptable and in line with the benefits that can be anticipated.

Specifying Functional Quality Indicators

The requirements must clearly state functionality quality indicators. That is, what things can someone look at, in the function of the application, that will clearly indicate good or poor quality? You may start with a business process flowchart that illustrates the business process and workflow that the application must implement or enable. However, don't leave things at such a high level. Specify other functional indicators, such as:

- Users should not have to re-enter data—once entered, data should be reusable across the application.
- Any screen displayed within the application must be printable in a format suitable for sharing with customers or management, as appropriate.
- Data entry must be enabled via the keyboard or via barcode scanning for all entry fields.

These types of quality metrics do two important things: they provide clear requirements that designers, developers, and testers can work with; they also reflect and communicate an understanding of current business problems. Reading a list of requirements such as this should remind the reader, "yeah, folks call the Help desk all the time now because they think the application locked up—I can see where displaying a progress bar will help alleviate that." In fact, when those reasons come to mind, they should be documented as part of the requirements so that the *purpose* and *reason* for the requirements isn't lost.

The Whys and Wherefores

Don't fall into the trap that legislators often do and simply specify requirements without specifying the reasons. In Nevada, one county's building code—clearly not updated in recent memory—states that a builder must replace the windows in a home if defects in the glass are "visible from twenty paces at high noon." High noon? Twenty paces? You might be inclined to simplify such a provision and say, "visible from 40 feet during daylight hours," but you'd be stripping the rule of much of its intended functionality simply because the rule doesn't document the *reason.*

The reason is that, in Nevada, "high noon" is when the sun is typically at its zenith for the day. When the sun is somewhat lower in the sky, say in the morning or afternoon, defects in windows can be more difficult to see because the light will strike them more in parallel and pass through more easily. At noon, light rays are more perpendicular to the plane of the window, so defects show more clearly. It isn't a question of "good light" as it is of the *right* light.

The twenty paces bit is just a little leftover Western charm.

Specifying Non-Functional Metrics

Here's where traditional methods of capturing requirements most frequently fail the business: Specifying quality metrics that *don't* relate to what the application does but which are nonetheless things people will complain about later if they're done wrong. The major areas tend to relate to security and performance. For example:

- Data lookups must never take longer than 5 seconds to return the first portion of the results.
- Security must be designed so that permissions are assigned to job titles, and users simply assigned to the correct job title within the application.
- The application must not print any screen or report that contains information identified as Confidential, unless the user selects a separate override option and acknowledges the risk.
- The time to complete the end-of-quarter balancing must not exceed 10 minutes once data entry is complete. Data entry must not exceed 5 minutes for a skilled data entry clerk.

Sometimes the line between functional and non-functional can be blurry; in an organization burdened with regulatory or industry compliance issues, for example, security requirements may well be seen as "functional." That's fine, and you shouldn't worry much about the distinction between the two. The point is to make sure you're considering these non-functional things, and stating requirements in a way that can be clearly measured. Any reasonable person on the project should be able to look at the application and objectively determine whether it meets the requirements; any collection of 20 or more people should be able to agree on whether the application meets the requirements. That means not using subjective terms such as "fast" or "small" but rather using absolute measurements.

Specifying Maintenance Requirements

How will the application be maintained in the long term? Have you given any thought to it? Your requirements should clearly state maintenance requirements because you certainly don't want designers and developers making assumptions about such an important piece of the application. Consider:

- How long you'll allow the application to be down for maintenance. A few hours each day? Several hours weekly? Not at all? The application needs to be engineered for the appropriate uptime, and the rest of its maintenance routines need to be built around this uptime expectation.
- How will data be archived? How will archived data be accessed? How much live data will the application be expected to deal with in a month? A year? Five years?
- How much data loss is tolerable in the event of a failure? A few hours' worth? Days? Minutes? How long should it take to recover from a failure? Is there a difference between the time it takes to get the application running and the time it takes to recover data?
- What should the process for deploying application updates look like? Must it be automated? How often will it occur? Will there be regular monthly releases? Quarterly? Or only as-needed? How will users know that an update is available? Will old and new versions be allowed to coexist?

Start with an acknowledgment that this application will be a living, evolving thing. That means errors will occur, things will break, and the application will change over time. Decide up front what the requirements are for dealing with these realities.

Specifying Use Cases

Finally, your requirements documents should conclude—or even start—with several use cases. Identify everyone who will use the application for any purpose, whether as an end user, a manager, an IT technician, or whatever. Describe what each person will do with the application, what data they'll provide to the application, and what they expect the application to provide them. Tell a short story about how the application will be used, what it will do during that use, and provide examples of the input and output that are expected. These use cases will be the very things that the application designer needs, the developers strive to enable, and the testers verify to ensure the application is doing what it's supposed to.

Management in the Driver's Seat

The idea behind a detailed requirements list is to put management, not programmers, in control of the application. By clearly specifying in objective terms what the application is providing, *why* it's providing those things, and *how* it is expected to provide them, you're on the right path to actually getting those things—on the right path, that is, to quality application delivery.

Design: Building Quality In

The design process picks up where the requirements definition stage ends. Requirements should be filled with business terms; the application designer's job is to translate those into a technical design capable of meeting the requirements. At a high level, this includes a basic application architecture as well as the specification of key technologies. The design phase is where you get your first look at the real cost of your requirements, and management has the opportunity to re-state the requirements in order to achieve cost goals, if desired. For example, if the data-transmission requirements necessitate the purchase of special database drivers, management can decide whether the cost is worth it. It's important that, if requirements are going to be re-stated, the reasons are also documented so that everyone is clear later on why the quality might be lower than originally envisioned. It's *fine* to knowingly accept lesser quality as a business decision; it's *not fine* to acknowledge the decision and make it part of the record. Chapter 6 dives into these design details in much more depth, but let's quickly overview the major components of design with relation to quality.

Designing to the Requirements

The designer should translate each of the requirements into unambiguous, technical designs; literally, drawings and diagrams, whenever possible. User experiences should be documented in flowcharts, and critical user interfaces mocked up, leaving little room for imagination and error on the part of the developers. The developers have the difficult task of making it all *work;* they shouldn't be expected to figure out what everything will *look like,* too.

Report mockups, indicating the data the reports should contain, should be part of the requirements; the designer can use those mockups to help design the flow of data through the application to ensure the developers have everything they need to produce the required reports.

The real key is to continually ask, "Does this element of the design not only meet the stated requirement but also meet the *reason* for the stated requirement?" If the requirement merely states that an order must be entered in 5 minutes, that doesn't give the designer much flexibility. However, if the reason for that requirement is to "free up salespeople's time," maybe the designer can realize that some of the data could be entered by another user of the application in a related process. That *reason* helps to streamline the way the application comes together.

Designing Quality Points from the Beginning

The design must also include quality points. As the designer specifies technologies, user interface elements, and application components, metrics for these should be stated. If a mocked-up user interface screen contains 50 input fields and is expected to be completed in 10 seconds, that may not be reasonable. Right there, we know that the design and requirements aren't in sync. If the screen is adjusted to contain 5 fields to be completed in 10 seconds, which seems more doable, that metric can be used during the development process to determine whether it's actually possible.

Designing for Code Quality

Designers also have to take into account how developers will implement the design, and create a design that lends itself to quality coding practices. Creating clearly defined application modules, or components, with clearly defined input and output expectations, helps place clear boundaries around what each developer must produce.

Designing for Performance

Designers must identify key points within the application where performance will be measured. It's unreasonable to expect a team of developers to collectively meet high-level performance requirements: One developer will state that "his" bit of code is as fast as it can be, and that someone else will have to "make up" the time. The designer alleviates this problem by creating more granular performance requirements, each of which must, of course, be reasonable and achievable, and documenting them. This allows fine-leveled unit testing to determine whether individual components are running properly, and gives each individual developer a performance deliverable.

Designing for Maintenance

Maintenance must be designed, as well, starting from the maintenance requirements. Work with the people who will be performing the maintenance to understand any other constraints they may be under and to produce a design that is capable of the level of maintenance required. Push back at the requirements team if the design is becoming overly complex or seems like it will be expensive to implement; allow the business managers to weigh the balance between what they want and what it will cost.

Designing for Users

Mock up user processes, task flows, and interfaces and present them to users. Ask them to validate the design before coding even begins, and ask them whether the flows and interfaces shown can be used to accomplish the use cases described in the requirements document. This "field test" of the design can be an invaluable way of discovering "gotchas" that weren't known to the requirements authors. For example, you might find that asking for a customer ID number on a particular screen isn't viable because the customer ID wouldn't be known at that point in the process. A redesign of the interface and even the entire task flow might be required, but doing so in the design phase will be vastly less expensive than a redesign after code has been written.

Designing Testing

Design test scenarios based on use cases and performance criteria. Describe what test data should look like; too often, poor test data is used, resulting in a lot of software defects making it through testing. Design a *means* of testing: You want testing to be happening throughout the life cycle, not near the end, so you might need to design "test harnesses" and other "throwaway" code that can be used to implement testing of individual or incomplete components.

Development: Quality Practices

Here's an area where organizations typically have little trouble attributing quality problems! Quality is often seen as being strictly under the purview of developers, but as we've seen so far, developers are really only responsible for code defects and, to a degree, performance, neither of which comprise the whole quality picture.

> **Cross Reference**
>
> Chapter 7 covers four major areas of software development that contribute directly to overall application quality.

Testing During Development

Most developers are accustomed to basic "unit" tests as they develop, although you always have to be watchful for developers for whom the phrase "successful compile" also means "fully tested." Developers who are working on standalone or independent components of an application should be held to a high standard in terms of unit testing: The designer should specify a complete set of inputs and outputs and the developer can ensure that all the specified inputs result in the specified outputs. The developer is also responsible for ensuring that the various workflows involving his or her component also work properly. Testing should be continuous, throughout the development process, and the results of those tests should be clearly documented for later review.

Developers should also perform performance unit testing, using the performance criteria specified in the design. Whether it's a performance metric for a particular task within the code or a metric for an entire module or component, having each developer code to objective performance metrics is the only way to ensure that the entire finished application meets its performance goals.

Managing Test Data

Test data is a major portion of the testing process. When developers are left without formal test data, they rely on made-up data instead—names like "John Doe" and addresses like "xxxxxxx," for example. Providing a full set of valid test data ensures that developers' unit tests are catching more defects earlier in the project's life cycle. If your application will support many languages, be sure to provide multilingual test data. Specify known bad test data, too, to make sure developers' code is handling it properly. Specify data known to be out-of-bounds, containing illegal characters, and other problems so that developers have an opportunity to test the full range of data that will eventually be thrown at their code.

It can become difficult in some industries to deal with test data. For example, "real" customer data might be governed by privacy regulations, so creating subsets of production data makes it difficult to use. There are alternatives, however: A jumbling of customer data can be used to mismatch names and addresses, for example, creating realistic test data without compromising individual customer privacy. Test data can be purchased from specialty vendors for a variety of purposes, or specialized test data management products can create test data that doesn't violate any rules and it represents a full gamut of possible inputs. In other situations, test data may need to be sequestered for security, and programmers may need to turn their code over to authorized testers for testing in a specialized, secured test environment.

Managing Coding Practices

Basic coding practices—using source control, commenting code, naming conventions, SGD automated builds, and other best practices—must also be managed, typically through peer code reviews, managerial code reviews, and even dedicated code reviewers for larger projects. Poor coding practices are a leading cause of code defects, and eliminating them requires constant vigilance. Specific applications can also be used to help detect code quality problems, and in some cases, to even help fix those problems—useful when you're dealing with legacy code in an application revision project.

Developing to Design, Developing to Requirements

The beauty of having a highly detailed design that maps directly to business requirements is that it forces developers to meet those business requirements without necessarily knowing or understanding them. We have to acknowledge that developers aren't hired for their business acumen, and that they don't work full-time in the actual line of business. Developers' knowledge of specific business needs and individual business problems is going to be necessarily limited; that's why a good design is so important. Developers *can* code to a well-made design, and so long as the design maps directly back to those business requirements, then, by definition, everything the developer does will also map to those business requirements.

Of course, that does assume the developers are coding to the design, which is why the design must be absolutely objective and unambiguous, so that anyone with sufficient technical knowledge or skill can examine what the developer is doing, review the design, and determine whether the developer is in fact coding to the design.

But I Wrote the Spec!

I was once hired to work on a documentation project for a piece of software. After the documentation had been written, we all sat in a room and reviewed it. The documentation authors were present, as were the people who had tested all the steps they had documented for using the software. At one point, we'd finished reviewing a rather complicated procedure and everyone seemed happy. One fellow, who was on the team who'd produced the software, looked incredulous. "You can't all be seriously okay with this," he said. "The product doesn't work the way this says."

Everyone's eyes got big, and they started reviewing what they'd written. "No," one of the testers said, "I tested this procedure. The software works exactly as described."

"That's impossible," the product team member said. "I wrote the design spec on this and several of these steps are wrong."

"Well," I said, shrugging, "I guess the guy who actually wrote the code didn't read your spec."

It happens. Provided testing and validation is occurring during the development process, however, you should be able to catch mismatches between the design and the code.

If for any reason the developer does *need* to legitimately deviate from the design, take that as an opportunity to really review the situation. Deviations from design inherently mean deviations from the requirements, and that's where trouble begins and quality is in trouble. Was the design wrong? If so, change it, but make sure it still meets the requirement. Is the requirement simply not achievable? Then revisit it. Whatever ultimately happens, make sure you're documenting the final result, the decisions that were made that led to that result, and the *why* behind those decisions.

Functional Testing: Verifying Quality

Finally, we come to testing, traditionally the last stop on the quality train. Of course, testing should be occurring throughout development, and the "field testing" of design elements during the design phase is really a form of quality testing, too. Here, we're talking about *functional* testing, where the various bits of the application come together for the first time to perform as an integrated whole. This is the focus of Chapter 8, which includes three major areas.

Unit Testing

Unit testing is the idea of testing individual modules of software independently to make sure each one is doing what it's supposed to do. This should occur throughout the development process as developers are writing code, but it should also be a discrete step that occurs separately as well. A *unit* is the smallest testable portion of a program; with a good design, an application will be comprised of a great many smaller units that can be tested independently rather than a single giant, monolithic program that can only be tested as a whole.

The design should specify test cases for each unit, typically specifying test input data and indicating what the respective outputs should be. In most cases, individual units aren't designed to be *used* independently, so the design should also specify the creation of *test harnesses,* which allow individual units to be tested by themselves.

Unit testing is a valuable part of the overall testing process because it validates the proper function of smaller working pieces. If a defect is discovered, it's easily narrowed because only one unit is involved. If each unit tests without flaw but later functional testing exhibits problems, the problem comes from interactions between the units—in other words, the scope for potential trouble is smaller. Individual units can also be tested more thoroughly with less effort than an entire application because a unit has a limited range of function compared with the entire application. Unit tests thus become not only a way of verifying quality but also of making more thorough testing more efficient and practical.

The key to good unit testing is solid, high-quality test data spanning the entire range of data that the unit will ever see in production. Another key is consistent testing, which typically is achieved through the use of test automation suites. The thing developers hate the most is troubleshooting an intermittent bug; by testing each unit the same way every time, defects can be caught more easily. The developer can be given the exact test data that caused the flaw, and once the flaw is fixed, the test can be easily repeated to confirm that the defect was in fact corrected.

Major challenges to unit testing generally come from what I call "sync." It's rare that an application's design will stand unchanged throughout the application's life cycle; feedback and evolution are natural parts of the process. It's critical, though, that *all changes derive from the requirements, and then from the design.* That way, testing can be included in all changes. Test harnesses will need to be updated, test data modified accordingly, and so forth. Unit testing becomes problematic when these test elements aren't evolving at the same pace as the application itself.

Functional Testing

Also called *system testing,* functional testing is when all the independent components of the application come together and are tested as a whole. Functional testing should absolutely be accomplished using good, realistic test data, automated testing tools, and other elements used in unit testing. In functional testing, however, the primary concern is completing the use cases that were part of both the requirements and the design. This is the time to make sure that the entire application not only works but does what it was designed to do, and that it solves the business problems it was originally intended to solve.

Functional testing breaks down into an almost bewildering array of sub-categories:

- GUI testing
- Usability testing
- Compatibility testing
- Error-handling testing
- User help testing
- Security testing
- Sanity testing (does it just do anything that seems crazy or unreasonable?)
- Ad-hoc testing (randomly "poking at" the application in an attempt to break it by performing unexpected actions)
- Recovery testing
- Maintenance testing
- Accessibility testing (if part of the original requirements)

Rethinking Testing

Testing is also the time to dig out the original requirements document and *test to it.* This is the time to make sure that the originally stated requirements are being met. Ignore the design; if there's some conflict between the design (and thus what the developers created) and the original requirements, now's a good time to find it and make an informed decision about what to do.

And the time to do this testing is *always.* Don't wait until the end of the project to discover that the code doesn't meet the requirements—that's not delivering quality. Begin rethinking testing and testing to requirements as soon as the first testing is capable of being done. Ideally, functional testing can begin very soon after the requirements phase, concurrent with development.

Performance Testing: Quality in Metrics

In Chapter 9, we'll look at an important part of non-functional testing: performance. Although performance can encompass many things, including scalability, capacity, response times, and more, we'll focus on two major categories: load testing and overall performance testing. These are both foundation topics that directly enable higher-level goals such as scalability and capacity.

Load Testing

Load testing is the idea of throwing an expected real-world load at an application to see how it performs. For slightly different reasons, you might also throw higher-than-expected loads at an application to see how large a load it can maintain, and to measure its performance at various load levels. The purpose of load testing is twofold: to ensure that the application, as a whole, can perform as required under the anticipated actual load; and to predict how much additional load the application can handle and what drop-off in performance can be expected at certain load capacities.

Load testing is typically pretty difficult to do accurately, even using automated test suites. Simulating the activity of hundreds or thousands of users can be difficult simply because of computer hardware bottlenecks. A single computer cannot, for example, output the combined network traffic of one hundred individual computers. True high-end load testing often requires a great deal of computer hardware, and some companies simply can't make that kind of investment.

Load testing at a unit level is typically easier, and it's one reason that load testing can be more usefully applied in multi-tier application architectures. Simulating thousands of database connections to a database server, or thousands of connections to a middle-tier component, is much easier than simulating thousands of users using a graphical user interface. Client software typically doesn't require load testing because it will only ever be used by one user at a time, spread across multiple computers; middle- and back-tier components, which are utilized by many user connections at once, are where load testing really becomes important.

The primary importance of load testing comes from the fact that software performance doesn't change evenly as load scales. For example, an application that has a 5 second response time with 1000 users will not necessarily have a 10 second response time with 2000 users. Instead, applications tend to perform well up to a certain point, at which time performance quickly begins to dip and then plummet. Load testing is designed to ensure that the "dip point" in performance doesn't happen while the application is under the anticipated production load.

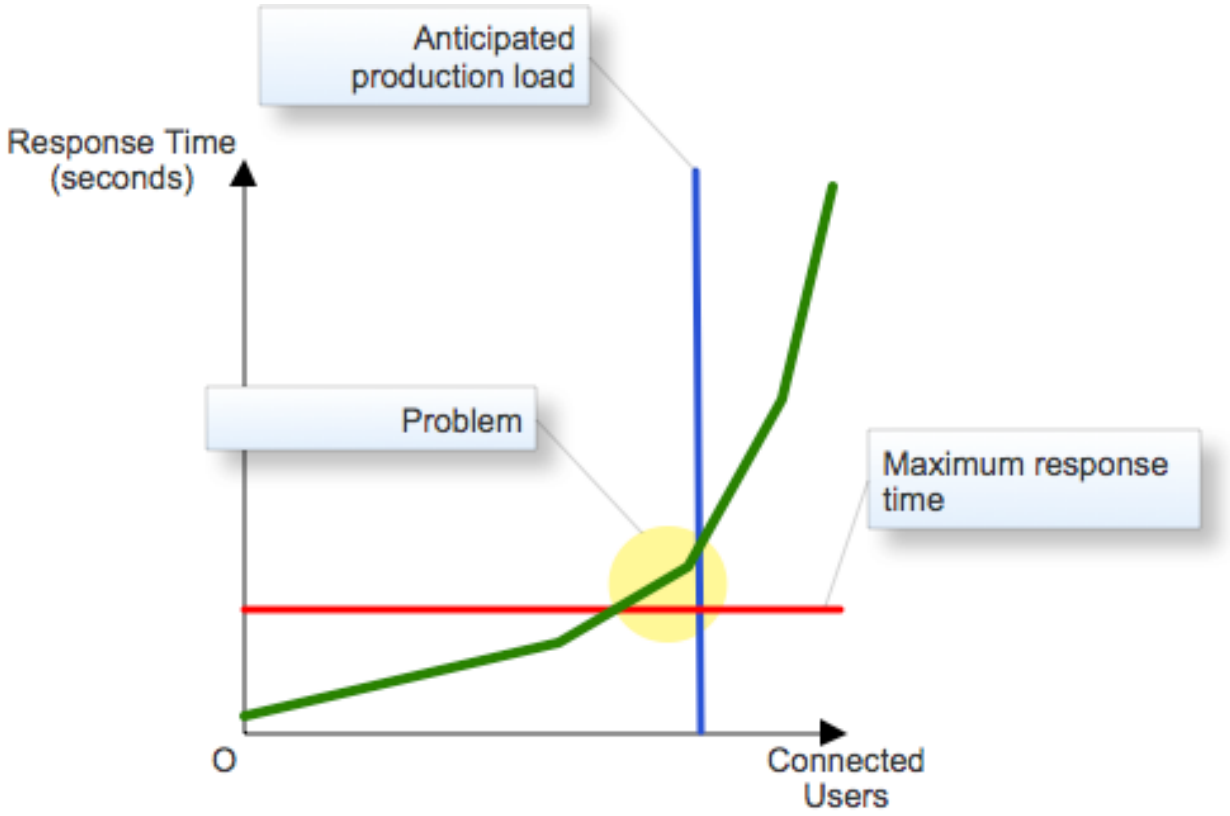

Figure 1.2: Example of a load testing problem.

Figure 1.2 illustrates this: The horizontal axis represents user connections, while the vertical represents application response time. As the number of connections increases, so does the response time—but not in direct correlation. The blue vertical line represents the anticipated number of production users while the red horizontal line is the maximum response time specified in the application's requirements. As you can see, a problem exists where the response time exceeds this maximum well before the anticipated production load is actually achieved. This is the exact type of problem that load testing is designed to spot.

Performance Testing and Tuning

In addition to overall load testing, general performance testing should be conducted throughout the development process. Individual units should be tested for performance whenever possible, according to the performance metrics detailed in the design. Functional testing of specific use cases should also be conducted, checking specific tasks against the performance metrics specified in the original requirements.

Some developers prefer to work on functionality (for example, getting it working) and then focus on performance later. I don't prefer that approach myself; I like to get the performance testing in nice and early for a very good reason: If the approach I'm using in my code isn't performing well, then I'm only going to get so far by tweaking it. I may need to take an entirely different tack in order to meet my performance metric, and that means I'm going to have to rip apart code. There's little point in "getting everything working" only to find out you have to tear it apart for performance reasons. Also, by testing for performance earlier, I can more easily determine whether the design or requirements performance metrics are achievable. If I can find out early enough that I'm never going to be able to hit my performance numbers, the designer and businesspeople can sit down with the design and requirements and decide whether they want to spend more money and time for better performance or compromise on the performance. At that time, I can also have a more realistic expectation about the performance I *can* achieve with the code so that the designer and business people know exactly how much performance they might have to compromise.

Quality Application Delivery

Finally, in Chapter 10 we'll look at the entire picture of quality application delivery. I'll take all the theory and procedures covered in the previous chapters and put them together into an actionable list that you can use to start improving your quality level. I'll summarize shopping lists for things such as tool sets that can help improve various aspects of quality, provide you with some report cards that you can use to assess your internal quality efforts, and more. We'll come up from the deep-dive into various aspects of quality and bring it all back together with a high-level, big-picture look at how all those pieces fit together and create an interdependent "supply chain" for quality. Take Chapter 10 as your final "to do list" for implementing and improving application quality in your organization, without wasting time and money.

Up Next: Defining Quality

This chapter was intended to give you a broad, high-level overview of everything that contributes to quality application delivery. In the next chapter, we'll start digging deeper by looking at things like the bad quality life cycle—how bad applications come to be, and how one bad application can set off an ever-deeper spiral of poor quality inside an organization. We'll look at traditional ways of measuring quality and decide what does and doesn't work, and take a fresh look at how business people outside IT tend to view quality—and how they perhaps *should* be viewing quality, instead. Finally, we'll spend just a bit of time really outlining the phases of the software development life cycle, and talking about how its iterative nature can create real issues for quality if you're not paying attention.

Download Additional eBooks from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this eBook to be informative, we encourage you to download more of our industry-leading technology eBooks and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.