Realtime publishers

The Definitive Guide™ To

Quality Application Delivery

sponsored by

MICRO FOCUS®
Leading the Evolution™

Don Jones

Copyright Statement

Chapter 2: What Is Quality, and Who Cares?

What *is* application quality? Before we go any further in discussing how to achieve it, we'd better agree on what "it" is. Plenty of people think quality is an intangible, something that's difficult to define and measure because it goes beyond the measurable; others think that quality is completely measurable, if difficult to achieve. Both are right.

It's sometimes difficult to see this in action in your own environment, so to help illustrate what quality is, who cares about it, and where it goes missing in a project I'm going to share from my own experience. I've worked in a number of organizations that had major software development projects that often suffered from fairly common quality challenges; we'll use those to home in on where the problems started and how they could have been avoided. Later chapters in this guide will dive deeper into specific topics.

Why Everyone Hates Applications: The Bad-Quality Cycle

Quality *is* something you can measure accurately; it's also something difficult to measure, involving users' perceptions, managers' perceptions, and a general, sometimes vague "feeling" about an application's quality. Any discussion of quality must therefore acknowledge these two viewpoints, attempt to understand them, and attempt to adopt a more holistic view of quality that incorporates both the measurable and the perceptive. With that said, what *is* the "Bad-Quality Cycle?" Why is application quality so challenging?

Does This Scenario Sound Familiar?

I've been peripherally involved with a number of major software development efforts—often as an IT administrator who would eventually deploy and support the application. Throughout this chapter, I'll share some of my experiences to help illustrate key points; I want to start with a sort of composite story that involves aspects from several projects at one company I worked for—truly illustrating the "Bad-Quality Cycle" that I'm sure you're familiar with.

We would start by seeing a need for an application—perhaps something to meet a new business need that had evolved, or perhaps something that replaced a commercial application we'd been using but that wasn't meeting our needs precisely enough. We'd start by outlining the requirements, but oddly (at least in hindsight; at the time it probably didn't seem odd), our developers and development managers spent relatively little time talking to end users or even business leaders within the organization. In most cases, they had a couple of big "planning meetings" about the new application, gathered a "wish list" of features from users, and then began designing the application.

"Design" usually meant a big entity relationship diagram (ERD) posted on the wall in the development manager's office. They were big fans of database design, and would spend weeks laying out what the database would look like, normalizing the design, and so forth. Then they'd start UI design, planning specific UIs that would map to the various bits of the database. Finally, they'd design the middle-tier components, which is where the business logic would live. In smaller applications, the business logic would be embedded in the UI elements, but the logic was usually designed last, as a sort of "connector" between what users would see and what would live in the database.

Next, we'd do some testing. *Lots* of testing, typically with dedicated testers and maybe a volunteer from the audience who'd be using the application—but not usually. Generally, it was just testers, many of whom were also developers. We felt that having developers test each others' work was efficient in terms of time and money as well as in catching bugs. Sometimes managers would make a contest of it, keeping track of bugs caught and giving a prize to whoever caught the most each month or quarter.

Next came the pilot deployment, where a lucky user or department would start using the app—usually in parallel with whatever they'd already been using. They would report more bugs, and start building a new "wish list" of features they would like to eventually see.

Last up was the final release, where the new application was deployed to the entire organization. This is where the nightmares began: Users reporting bugs we'd never found, users complaining that the application forced them to change the way they worked—and not in a beneficial manner—and more. Everyone *hated* the application. The Help desk would be flooded with calls for assistance, we'd be scrambling around trying to back up and restore data, and inevitably an executive would start pondering whether to pull the app and go back to the old way.

That potential decision set off alarm bells in the development side of our IT department. They quickly began doing triage on the complaints, taking each one as a requirement for change. A new build of the software quickly entered the development cycle, designed to address all the bugs and complaints we'd heard about. A new release would sometimes be developed in just a few weeks…with more problems. We'd broken six things in our efforts to fix a couple of bugs, changed the workflow that users were *just* starting to get used to, still weren't producing the reports that managers needed to see, and so on. Too deeply committed to pull back entirely, executives would start glaring at our CTO and development managers, who would gather up all the Help desk complaints and start hurriedly working on yet another revision designed to address *this* cycle of issues.

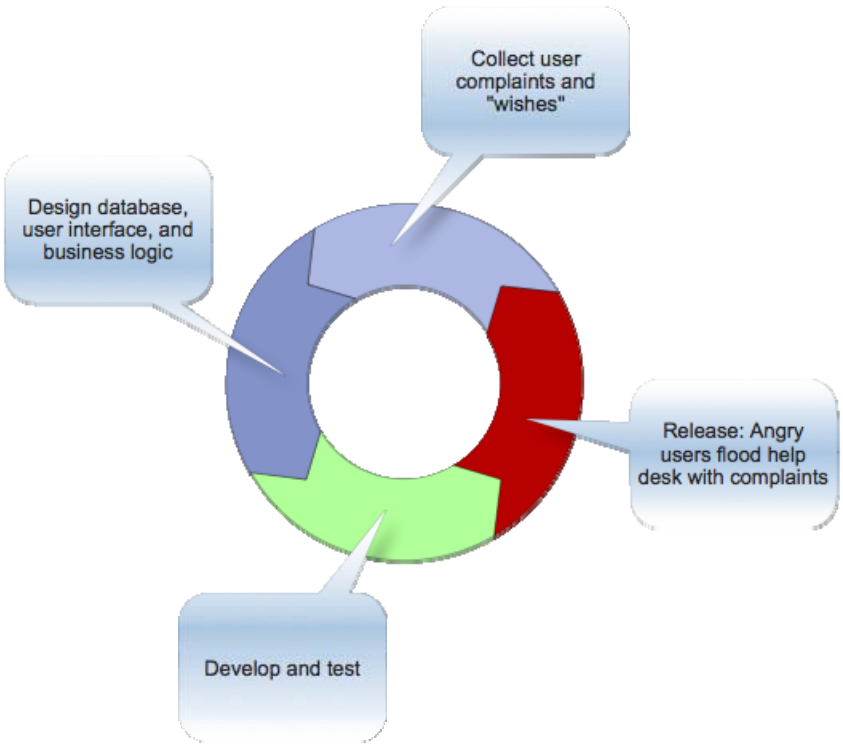

Figure 2.1: The Bad-Quality Cycle: complaint- and reaction-driven development.

You see the problem? We weren't working on an application; we were working on a train wreck that was becoming worse by the moment. Our entire process treated "quality" as a measure of bugs found, wasn't driven by real business requirements, and did nothing but create applications that everyone disliked; ultimately, it all conspired to create a never-ending chain of bad new releases and ulcers for everyone on the development team. Sadly, this is one of the most common scenarios in the IT world: internal applications that continually cycle through the "Bad-Quality Cycle." Why?

Poorly Defined Requirements

Our problem began with poorly defined requirements. We didn't spend time determining what the *business* needed; we simply asked users and managers for a "wish list" of features, assuming that *their* needs were the same as the *business'* needs. Not only was such not the case but collecting "requirements" in that fashion never resulted in a *complete* set of requirements. We were always missing something, often major items.

User Requirements ≠ Business Requirements

In designing a retail point of sale (POS) system, several users of a previous system said that they needed to be able to manually enter credit card numbers when a card wouldn't swipe for some reason. It sounded reasonable to our developers, and it was added to the software they were developing. It wasn't until months after the software was in widespread production use that our Loss Prevention Specialist came raging into the IT department screaming about this "bug." Our merchant banks gave us excellent discount rates on credit card charges because we *only* accepted swipe-able cards; since the introduction of our new "feature," our processing rates had nearly doubled and we'd found at least five instances of credit card fraud, where employees were charging to card numbers they'd stolen. This is just one example of how asking users for "wish lists" can seem beneficial but doesn't necessarily constitute "collecting business requirements." It also illustrates the need to get *everyone* affected by an application involved in defining requirements.

We were, we discovered, exceptionally bad at collecting business requirements. Unfortunately, we were so bad at it that we didn't even see how a lack of good business requirements is exactly what doomed the rest of our entire software development process.

In one memorable project, we made a valiant effort to do a better job of defining our requirements up front. Our intentions were good, but without experience or tools to help us, we were easily sidetracked. It turns out that "requirements" can often be a hindrance as much as a help:

- Formalized requirements from elsewhere in the business are the best way to achieve solid requirements for software. If you have existing service level agreements (SLAs), customer commitments, or other hard, defined requirements, those will translate easily to software requirements.
- Informal requirements—what you can think of as "traditions" within the organization—can be difficult to capture. These requirements can include factors such as a longer-than-normal workday, meaning maintenance windows might be smaller. It's important to capture these, as they contribute significantly to the software design.

Note

Not all requirements are useful and beneficial, even though they might seem so. Always ask yourself how a requirement is contributing to the business' bottom line; don't be tempted to load a project down with so many "good ideas" that it simply becomes impractical.

Design Without Perspective

Lacking any real business requirements, our designers went straight for database design, something they were taught in college as crucial to the success of any application. *Technical* success, perhaps, but not *business* success, as we had ample reason to discover. Our designs never worked especially well for our business:

- UIs were often poorly laid out, often forcing users into a workflow that didn't even remotely match the way their jobs actually worked. One system, for example, asked users to enter data on one screen that wouldn't even be created until later in the process—because that's the way the database had been designed.
- Performance, from the users' perspective, was often poor. How often have you called a company on the phone and waited while the agent apologized and said, "The computer is slow today?"
- Reports often simply parroted what was in the database, meaning managers would have to look at three or four different reports in order to get the information they needed.

Database Design ≠ Efficient Software

One project we worked on was for a real estate database. Our designers, being designers, sought to create a highly normalized database design, which resulted in a single street address using no less than *nine* tables: One for pre-address directions (N, SW, and so on), another for the address number, another for the post-address directions, another for the street name, another for the street type (Rd, Ave, St, and so on), and so forth. A beautiful database design, but a dead dog in terms of performance. Looking up an address would often take several seconds at high system load, simply because the data was distributed in so many different places.

This is another example of a design not being guided by business requirements. We had never stated any performance goals or considerations in the requirements, so the designers simply did the best design they could—which was a good *technical* design but wasn't very *practical* for our needs.

Programmers as Psychics

No application design has ever been, or will ever be, created that fully addresses every question a programmer will come across when building that application. Ideally, however, when programmers run across a question, they have a list of business requirements for the application, and they can refer to that list to help guide their answer. Without a clear set of business requirements, though, programmers have to guess what the business would want them to do—and they usually guess wrong, simply because few programmers have a good business perspective. They're not hired for their business acumen; they're hired for their programming expertise. That typically results in technically adequate decisions, but rarely results in the right thing for the business. This reality points right back to the source—sound business requirements. Without them, the entire project isn't driven to business needs. It's driven to programmers' preferences.

Programmer Preferences ≠ Good Software

Back to that POS system: Another major problem we had with credit cards resulted from, again, a lack of business requirements and a reliance on programmers' "intuition." Our POS system, like most, produced printed receipts. Like many PC-based POS solutions, the content of the register receipt was also displayed on the screen. Our programmers had the receipt include the full credit card number for credit card purchases; nobody had told them to do anything differently, and they felt that having the full number would be a good troubleshooting tool as well as a form of data backup.

Unfortunately, the screen display didn't clear at the end of the transaction—meaning the last credit card used by a customer sat on the screen until a new transaction was started. This didn't do much for improving our security or compliance standings, and created real problems when some of those credit card numbers were copied and used for unauthorized purchases.

Had system security been a business requirement, developers could have referred back to that and realized that the entire number should never have been displayed. Without such a requirement, they took a very common and understandable technical approach, which ultimately damaged the business.

Haphazard Testing as a Sole Quality Checkpoint

Aside from having no business requirements firmly in place and clearly communicated, our other major problem was that "quality" wasn't a real concern until it came to testing. Bugs were our only measure of quality: No bugs = great quality. The development managers' bug-hunting contests were the perfect expression of this mentality.

The problem is that a completely bug-free application is *useless*, or *worse* than useless, if it doesn't meet the business needs. Our testing was never driven by "how well does this application meet the needs of the business," but rather by "how often will the program crash due to a bug?" That meant *fixing* bugs was paramount, even if doing so resulted in changes to the application that actually made it *worse* for the business! Our testers and developers had no list of requirements they could look at to help them understand that a given feature *had* to work in a certain way, even if that meant a tougher programming task for them. In many cases, features were simplified, modified, or eliminated to make programming easier or to eliminate the possibility of a bug—regardless of whether that feature, as originally designed, was crucial to the business.

Bug Free ≠ Useful

My favorite example relates to a system that was designed to push written communications from our corporate office out to all of our branches. The corporate office composed messages on an AS/400 computer, which uses the EBCDIC character-encoding system rather than the more common ASCII system used by our branches' PCs. The system that pushed the communications, then, had to translate between the two character-encoding systems. A bug in the translator module incorrectly translated the exclamation mark (!) into an unprintable character, which caused the branch PCs to crash. The programmers' solution was to simply forbid the use of the exclamation point character in messages.

Once again, without business requirements to drive each phase of the process, our developers simply took an expedient, technically acceptable solution, which ultimately did nothing to help the business. You should have heard the Help desk calls when users tried to compose messages without using a common punctuation character!

So how did all these wrongs occur? What was wrong with our process that we couldn't produce an application that everyone agreed was high quality? When our developers released software, they were generally proud of it—why didn't anyone else agree with their quality assessment?

Traditional Quality Indicators

Ultimately, the problem came from our use of traditional indicators of quality. "Quality" is intangible; you can't point at something and definitively declare it to be of a given level of quality. Instead, you rely on certain metrics.

With a car, for example, you might derive your quality assessment from how solidly the doors close, how flush the trunk is when closed, and whether it seemed to be solidly constructed from good materials. In the long run, though, your quality assessment might be colored by frequent breakdowns, poor fuel economy, or irritating controls on the radio. In other words, your initial quality indicators weren't comprehensive: Additional factors became important to you over time, and what you once thought of as a high-quality car turned out to be a real clunker.

Traditionally, application development has had a similar, lopsided view of quality—one based almost entirely on software defects (bugs). An application is "high quality" when it has no bugs, period. Companies invest tens of thousands of dollars in "defect tracking systems" (bug logs), in training developers in coding practices that help avoid bugs, in static code analysis tools that help spot common bugs, and so forth. To be clear, *that money is well spent,* because being bug-free *is* a *part* of a high-quality software application. The problem is in assuming that the bug count is the *sole* measure of quality. It isn't.

Perform a Web search on "application quality" and you'll be confronted with phrases like:

> What is the one activity or phase that improves the quality of your application? The answer is an easy one: Testing, and plenty of it.

and

> Dramatically reduce the time, cost, and complexity associated with building, deploying, and maintaining Web sites. [Our] content quality solutions provide "out-of-the-box" testing and reporting….

Although these factors definitely contribute to quality, they certainly aren't the entirety of it nor are they even the beginning of it. One software architect I worked with said that, "quality is a measure of excellence," which sounds great but doesn't really add anything concrete to the definition.

In a 2006 article (Source: http://www.theserverside.com/tt/articles/article.tss?l=ArchitectAppQuality), software architect Allen Stoker wrote:

> While quality is often directly associated to defects, it is typically not something that can be achieved by implementing a few test cases. In most situations it has deep roots in architecture and process. Many would say that these are big project issues and excessive rigor is not needed for a simple utility program, but I would challenge that assertion. It is difficult to develop complete and accurate documentation of requirements, and design, and code, and test cases, and build environments, and, and, and... I just want to get this code done! Don't fool yourself into this trap. While you may choose to skip any or all of these steps they still apply to the smallest of projects. It's easy to justify the lack of a formal design document because you have it all in your head, but you may find the details escape you six months later when you need to make a change.

He summarizes with a concise definition of quality:

> I do believe that there is a universal understanding of quality. People want to use applications that do things right and don't break…Quality begins in the team - not the application. Proper planning, communication and processes are essential to any successful project. Projects that lack these fundamentals will likely produce problematic applications.

Exactly. The "don't break" part, of course, refers to bugs; the "do things right" part is something entirely different, and something not measured by traditional quality indicators.

Let's start correcting quality misconceptions with a quick list of the expectations your users will have when it comes to quality, and see how some of the traditional quality metrics contribute to achieving those expectations:

- Doesn't crash
- Works the way users work
- Works quickly
- Produces the information the user needs
- Is secure
- Makes business processes easier to complete correctly
- Prevents erroneous data from entering the system
- Meets legal and/or industry requirements

Of all these varied expectations, the traditional measure of quality—an application that is bug-free—only helps meet one (doesn't crash). Clearly, in order to achieve a more universally accepted definition of quality, as in "applications that do things right and don't break," we need to add more activities.

Quality from a Business Perspective

The previous list represents a more common approach to quality from a *business* perspective: Does the software do things that we need, in the way we need them done? The business *cares* about bug-free applications, but most businesses accept the fact that no application is perfect the first time out; most are willing to accept bugs, which can be fixed, in exchange for an application that basically accomplishes something the business requires.

Although "bug free" is an excellent technical measure of an application's quality, much like a car that never breaks down, businesses have a much broader set of criteria. You wouldn't, for example, want to buy a car that would break down only rarely but that was difficult to operate, uncomfortable to sit in, and only made left-hand turns. Let's define some of the other things a business looks for in a "quality" application:

- **Doesn't crash.** Obviously an important consideration, but not just from the perspective of finding bugs. Applications also need to not crash even at the extent of their intended usage—meaning, for example, that an application intended for high-volume use must be able to run successfully at a high workload.
- **Works the way users work.** Businesses want applications to *enhance* business processes, and ideally make them easier to carry out. Applications should allow users to employ their existing workflow unless there is a business reason to change that workflow and enforce a new one.

- **Works quickly.** Applications need to have a high level of *perceived* performance. Users should rarely, if ever, be waiting on the application; users should never have to tell a customer, "Hang on, the computer is slow today." That said, performance is often one of the last considerations in an application—and therefore one of the most difficult things to address late in the process. Performance can be especially critical in the growing world of self-service applications, such as Web applications that are increasingly common for banking, insurance, and other applications. It can be difficult to define performance metrics in the beginning; we'll explore this task in later chapters.
- **Produces the information the user needs.** It seems obvious, but many applications don't, in fact, give their users the information they need, or they make users hunt around for the information in various windows and reports. Users need a given set of information for any given task, and they should be able to get most, if not all, of that task's information in one spot.
- **Is secure.** Security is rarely a concern in application development. However, if the business is subject to industry or legal requirements, the application needs to support those. Even companies that have only their own internal policies to deal with *need* some kind of security in an application.
- **Makes business processes easier to complete correctly.** An application must not only provide a familiar, efficient workflow for the user but should also make difficult or uncomfortable workflows—mandated by business requirements—easier to accomplish. Applications can do so by helping the user to fill in the right data, guiding them through the process in the right order, and so forth.
- **Prevents erroneous data from entering the system.** Data validation isn't just about preventing bugs (making sure users don't put names in a number field, for example) but also about ensuring that the business has complete, accurate data to work with. The business needs to define what "complete and accurate" looks like, of course, but it's up to the application to enforce it.
- **Meets legal and/or industry requirements.** Whether these requirements pertain to security and privacy, as already mentioned, or to operational procedures or other criteria, the application needs to help protect the business by helping users comply with the necessary requirements.

So, what activities can be added to help achieve this business perspective of quality?

Expectation	Activity			
	Eliminating Bugs (development practices and testing for defects)	Collecting and Managing Business Requirements	Designing to Meet Business Requirements	Programming and Testing to Determine Whether Business Requirements are Met
Doesn't crash	✓	✓	✓	✓
Works the way the users work		✓	✓	✓
Works quickly		✓	✓	✓
Produces the information the user needs		✓	✓	✓
Is secure		✓	✓	✓
Makes business processes easier to complete correctly		✓	✓	✓
Prevents erroneous data from entering the system		✓	✓	✓
Meets legal and/or industry requirements		✓	✓	✓

Table 2.1: User expectations and the required development activities needed to meet those expectations.

In other words, the overall software development life cycle, starting with the collection of real business requirements, can help meet many more quality expectations than the simple elimination of bugs. It's important that *every* step of that life cycle be aware of the business requirements for an application. Simply put, the business is paying for the application's development, thus, the business deserves to get a quality product—by the business' own definition of "quality"—in return. The key is to have a proper software development life cycle; one that is focused on quality during every step.

The Software Development Life Cycle

Let's examine the various stages of the software development process and look at how each one contributes to quality or a lack thereof.

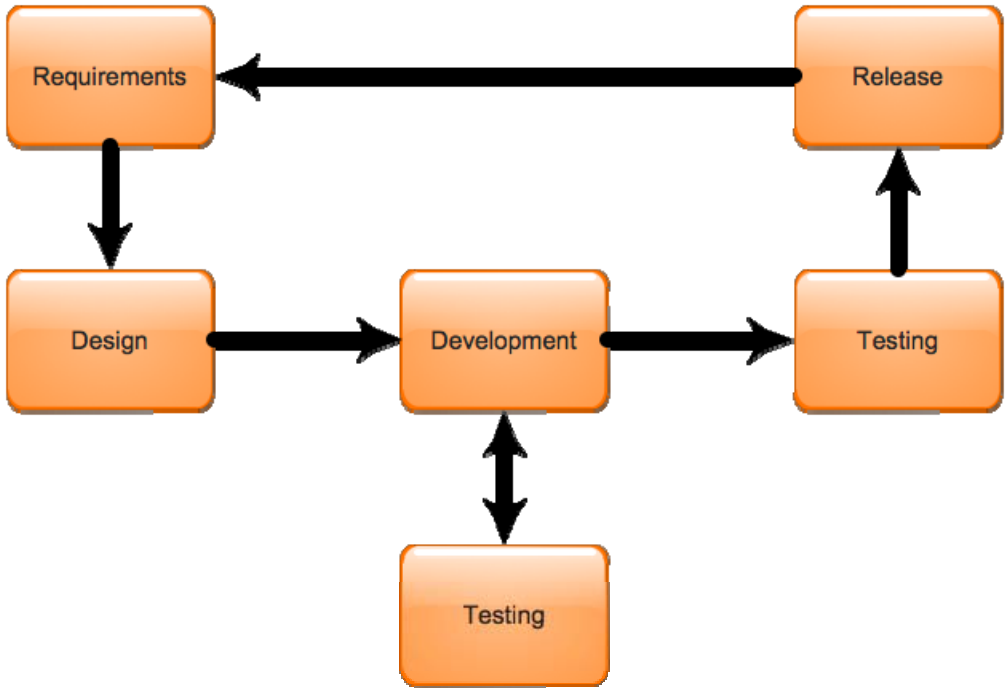

Figure 2.2: A typical software development life cycle.

Although different organizations and management frameworks define different discrete steps in the development life cycle, the diagram in Figure 2.2 reflects a common set of simplified steps. Let's examine each of these to see where they can contribute to quality, and where each of them often fails to do so.

Requirements

This first step in the development process is where the application's basic functional requirements are developed. What will the application do? How will it work? Who will use it? What will they need to be able to accomplish? What reports will the application produce and how will data be input into it? How many people will use it at once? When will its peak usage be during each day? How much data will it process at its peak usage?

Too often, organizations shorten or completely skip this critical step in the process, feeling that their in-house development team "already knows" what the application needs to do and how it needs to work. Sadly, nothing could be further from the truth: Few developers or even development managers have worked in every aspect of the organization, have dealt with the unique challenges each eventual application user will see, or fully understands the actual business processes and requirements that the application is intended to support. Interestingly, the requirements phase is where all quality begins, and in many organizations, it's the phase where the biggest and best opportunities to produce a quality application are lost forever.

Case Study

I used to work for a networking company in the US Mid-Atlantic area. We were an ISO-9001 certified company, which meant that all our business processes and methodologies were fully documented so that they could be repeatable and consistent. The ISO regularly audited us to make sure we were complying with our own processes and keeping those processes up-to-date. We figured we were as prepared as anyone could be to produce a custom, in-house application that helped implement our processes—after all, we knew *exactly* how the business worked because all the details were right there in our flowcharts.

It turns out that nothing could have been further from the truth. Over a year and a million dollars into development, we held a meeting with our senior developers, me, and our senior field management to discuss the progress of the application. That's when we realized that *nobody* really knew how our business worked, ISO flowcharts notwithstanding.

One regional director said that we needed to have Microsoft Visio drive our custom application: His engineers used it to create network diagrams complete with part numbers; those diagrams should be read to create purchasing documents and formal proposals. Another regional director immediately disagreed, saying that he didn't like to dedicate that much engineering time on a proposal, and suggested that Microsoft Access be used to drive the process of creating purchasing documents. A third said that she usually consulted with the networking equipment vendors, and they delivered plans in AutoCAD format, and that she needed our software to read that and construct purchasing lists and proposals. Needless to say, none of them had a high opinion of our application quality when they were told the software did none of those things—because none of those things were on our precious ISO flowcharts.

Our sin was in not spending more time talking to the business leaders who would be using our product, and in not taking the time to understand how they *really* did business. Had we simply had that same meeting a year (and a million dollars) earlier, we would have realized the inconsistencies in our business processes and set out to fix *that* before we even started designing UIs and writing code.

Close to 2 years and many dollars later, the project was ultimately cancelled without a single delivery to our users. It had been doomed from the start by a complete lack of real-world business requirements.

The requirements phase should produce a complete list of the application's ultimate requirements—a checklist, if you will, of everything the application will do. It should clearly identify the various audiences (users) to which the application must cater so that later development phases can speak to representatives of those audiences if clarification or expansion is necessary. Ultimately, the requirements phase should result in a simple (although often lengthy) document that says, "This is a quality application if, upon delivery, it performs the following functions…."

The requirements phase is also where the vague, perceptive view of quality is identified and accommodated. The requirements document must specify acceptable performance levels, from an end-user perspective, for key operations and functions. Ideally, the requirements document should take into account everything your users and managers have always hated about applications, and address those problems by providing clear, business-driven requirements that designers, developers, and testers can continually check against to make sure they're meeting the criteria.

Note

The requirements phase is akin to the part of building a new house where the new homeowner expresses their desires: "It needs to have a big living room so that my whole family can watch our big-screen television together, and it needs to have a big hot tub in the master bathroom for my wife to enjoy. I want upscale materials like travertine and granite throughout, and we prefer light-colored woods to dark ones. We love gourmet cooking, so we need high-end kitchen appliances. We don't entertain often, though, so we don't need a giant kitchen with a lot of wasted space."

At the end of the day, *only the requirements document* defines quality. If there is no such document, or if the document defines quality poorly or loosely, *then the lack of application quality is the fault of the business,* not the fault of the software developers, QA team, testers, designers, or anyone else. In other words, if the business leaders cannot define quality and communicate it in a document, they shouldn't be surprised when they don't get it.

Cross Reference

Chapter 5 will discuss the requirements phase in detail.

Design

Many organizations treat their design and requirements phase as one; for maximum quality, however, design should be an independent phase, as it involves fundamentally different skills. *Requirements* should be 100% business-driven, and the requirements document should come primarily from business, not technology, leaders and users. *Design* is the act of translating those business requirements into a technology-focused specification that programmers can implement.

> **Note**
>
> The design phase in new home construction is where the architect and engineer take the new homeowner's requirements and create detailed blueprints that the various building trades can follow to actually build the house.

The design phase carries a lot of responsibility. For example, the requirements document might specify that "users be able to enter new sales orders in 5 minutes or less," but the designer is responsible for actually finding a way to make that happen. The designer might need to conduct tests using prototype UIs to discover the best way to meet that 5-minute goal, allowing some users who will actually be using the interface to try various interface permutations until they hit on the most efficient one.

The designer also has a major contribution to the end quality of the application by designing in quality checkpoints. Although the requirements might state that, "retrieving an order must require no more than 10 seconds of waiting," the designer needs to break down that requirement into the discrete steps that the programmers will be taking, perhaps specifying maximum response times in the client application, in a middle-tier component, and from the back-end database. The designer also has the responsibility of designing an architecture that encourages quality coding practices, that scales as needed, and for designing maintenance capabilities as specified in the requirements. Many of these steps may require the designer to go back to the requirements team and ask for clarification, and it's therefore critical to have a designer who knows to do so rather than taking a guess. It's important that the entire design be driven first and foremost by the business requirements; if the requirements aren't covering a portion of the design, the designer *must* go back and ask for additional requirements to drive *any* design decisions that are made.

Finally, the designer should also design the testing procedures that will be used throughout the development life cycle, specifying test data sets, test cases, how tests will be executed, which components will be tested independently (and how), and so forth. These tests are absolutely crucial to ensuring the ultimate quality of the application, and by designing the test data *up front,* as part of the application design, programmers have accurate, representative data to work with to ensure that they're continuing to meet the original requirements of the application. Obtaining representative data can be difficult in some cases, as the data itself may be sensitive; that's something we'll explore more in a later chapter. As Figure 2.3 shows, the design impacts each and every portion of the application and affects everyone who develops and ultimately uses the application.

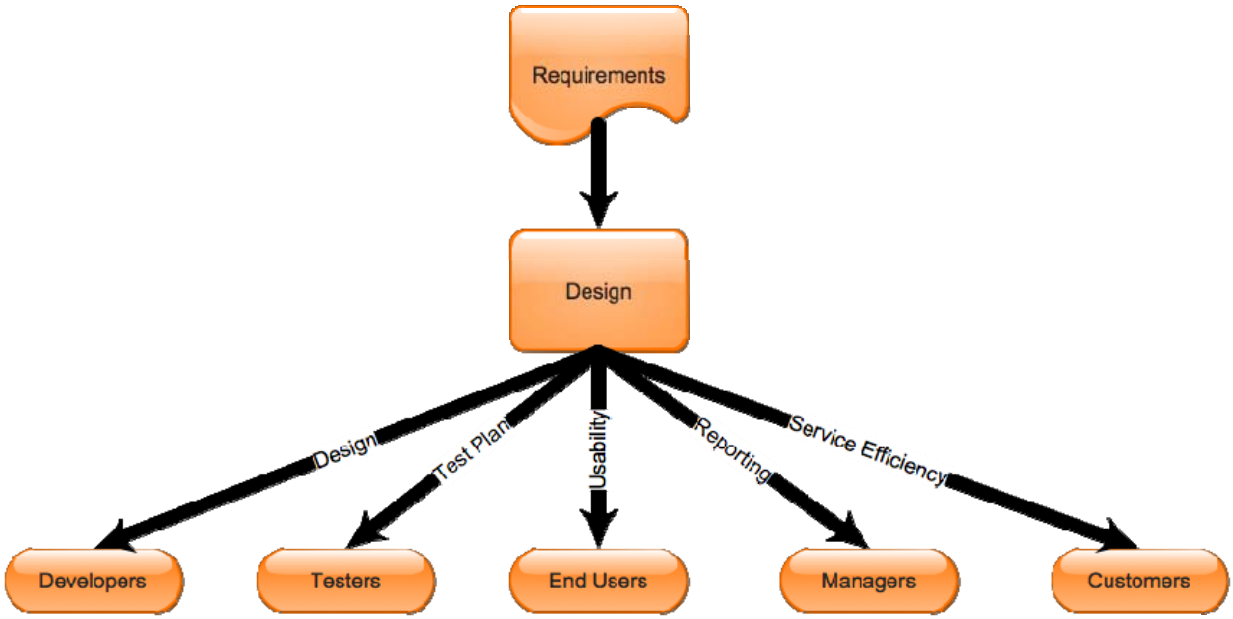

Figure 2.3: The design drives and impacts quality for every contributor to, and user of, the application.

Ultimately, the designer is responsible for ensuring whatever level of quality is specified in the requirements. A failure in the application design can lead to expensive, and sometimes irreversible, damage to the application's final quality, even so far as to impact the ultimate level of service provided to the business' customers.

> **Cross Reference**
> Chapter 6 will cover the design phase and its goals and pitfalls.

Development and Testing

Development and testing are where the majority of organizations first focus their quality efforts. After all, it's easy to look at a poorly made product and blame the workers who actually built it; issues such as poor code quality (in other words, "bugs") are felt to be the fault of programmers, as are issues such as poor performance, unusable UIs, badly designed reports, and so forth. In many cases, software programmers are "where the buck stops" in terms of carrying the blame for poor quality.

As we've seen, though, the requirements and the design actually carry *more* responsibility for quality; without a good set of requirements and a clear, requirements-driven design, programmers *can't help but fail.*

> **Note**
>
> The development phase is much like the actual construction phase of a new house: Without clear blueprints that were developed directly from the new homeowner's requirements, the construction crews aren't likely to produce a house that the customer is happy with.

Software programmers are kind of in a no-win situation when they aren't given clear, business-driven requirements and a clear, requirements-driven design. Programmers can do a great job of coding and still produce a lousy application that doesn't meet the business' needs, or they can do a poor job of programming and produce a lousy, buggy application that doesn't meet the business' needs. Without solid requirements and a good design, though, programmers can *never* be expected to produce a great application that meets the business' needs.

Testing—specifically, unit testing that focuses on relatively small sections of the application rather than the entire application as a whole—must be a part of the development process, and in many cases will be conducted by the developers themselves. Such testing *must be driven by the design,* which must specify unit testing requirements and goals, including performance metrics. Without such goals and requirements, programmers will simply test to make sure their bit of the application compiles correctly and runs without error—usually using contrived test data that (at least subconsciously, if not deliberately) avoids any known weak points in the code.

> **Note**
>
> Unit testing is a lot like the periodic inspections made as a house is being built. These are designed to catch problems such as improperly installed plumbing, bad framing practices, and so forth. But they're only a complement to another important unit test: Periodic inspections by the architect and new homeowner to make sure the home is not only *technically* correct but also being built in such a way that the design and requirements are being met.

It is the nature of developers to dislike testing, especially testing their own code—it's a bit like trying to copy edit something you've written: you tend to overlook problems because you're so familiar with what you're looking at. Unit testing is necessary at the development level, but it needs to have clear design goals—not just "bug free"—so that developers are unit testing to specific metrics. This is also one reason that formal testing is often conducted by testers, not developers; there is a natural fiction between the two groups as a result, but that's both desirable and beneficial to the overall quality of the final software.

Test Cases, Test Data, Test Quality

Testing is often held to be a major contributor to a quality application, and it's certainly the *last chance* for a quality check. We've seen, however, that it's far from the first point at which quality should be introduced to a project, and the quality of the testing itself will be driven by the design.

I used to work for a software consulting firm that produced several major applications each year. Typically, we worked from requirements given to us by a client, and then worked with the client to create the design. On one famously mismanaged project, we agreed to work from a set of requirements *and a design* provided by a customer who had a good amount of in-house development experience. Unfortunately, the client's design didn't address testing in the least.

Our developers did their best, testing each component they wrote to make sure they got the expected outputs when they provided given inputs. Unfortunately, their inputs were far from real world. In a simplistic example, one component—responsible for accepting new customers into a sales-entry system—was tested using customer names such as Smith and Jones. The minute we turned the application over to the client, and a name like O'Reilly hit the component, everything failed. The developer had never thought to test names that contained punctuation marks. The customer, understandably, thought this was obvious, but had he specified a sufficiently broad test plan in his design, we would have caught the problem in the beginning.

This problem then cascaded down through several connected components, middle-tier software, and even the backend database. Reports were affected, and that one apostrophe wound up costing about $7000 to fix—dollars the client was obligated to pay because his design didn't provide us with the test data that we normally would have spent time developing up front. By the way, our standard charge for designing test data was only about $2000, so the customer saved himself nothing by providing his own, inadequate design.

We keep coming back to the design and how the design itself must be driven by requirements.

Testing

Wait, *more* testing? Absolutely. Although developers can and should test individual units of code, at some point the entire application has to undergo final acceptance testing. This is an area where the design continues to be important, but as Figure 2.4 shows, the final phase of testing actually answers to a higher power—the requirements.

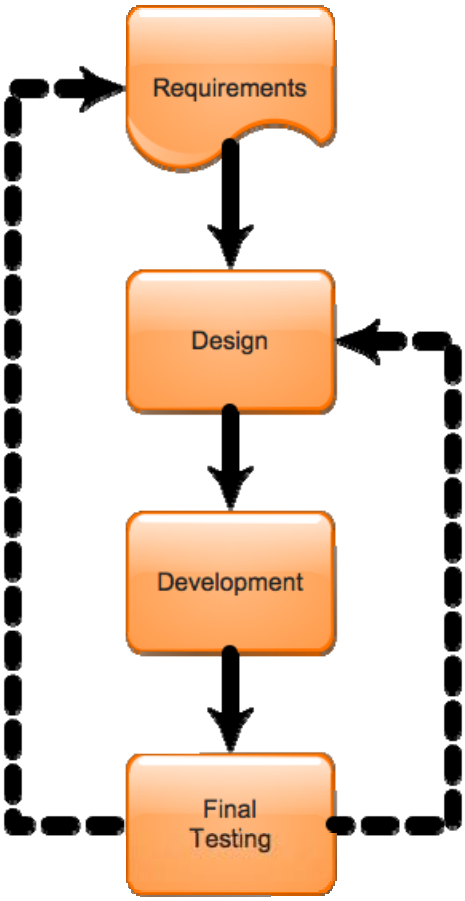

Figure 2.4: The final tests need to refer back to the requirements as a measure of success.

Although the final phase of testing must obviously be driven by the design, including test cases and test data that targets the entire application, the final testing must also refer back to the original business requirements to make sure that the final applications meets these requirements. Ultimately, unless every requirement can be checked off as "present," the application isn't a success.

This is a key element that many organizations miss: In organizations with a lower level of quality maturity (something we'll discuss in Chapter 4), the focus of testing is almost exclusively on finding and eliminating bugs. That's certainly a desirable activity, but a bug-free application is not necessarily a quality one. Does the application do what it's supposed to? Are its performance and usability metrics correct? Does the application provide the proper reports, and enable the required maintenance capabilities? This final test is literally the point where the requirements document becomes a checklist (so it's helpful if that document was designed as a kind of checklist in the first place) to determine whether the application will *meet the business goals.* Honestly, most businesses would rather release an application that has a few bugs but otherwise meets the business requirements than release a bug-free application that misses the business requirements to any significant degree. Bugs can be fixed, often easily; missing the business requirements indicates a flawed design.

> **Note**
>
> This final test is essentially the homeowner's final acceptance walkthrough: It's far too late to worry about (or even detect) most construction "bugs" at this point, with all the drywall in place and paint covering everything. Any technical defects—a crooked wall, mislaid tile, non-functioning light switch—can generally be fixed if they're found. But if the house doesn't meet the requirements—if the hot tub is too small or the kitchen appliances of inferior quality, it might be too late to fix them without massive expense and loss of time.

It's important that the final test *not* be the first time that the design and original requirements are considered; the design should have included sufficient detail and sufficient unit testing to ensure that the design and requirements were being adhered to all along. The final test is simply the *last chance* to catch missing requirements; fixing those at this point will likely be expensive.

Release

Last is the product's final release to its end users. This is another phase that is often given short shrift in many organizations, and it can contribute greatly to the ultimate quality of the application. How often have users had a new application unexpectedly dropped on them—and, when that happens, how often do users have a positive perception of the application? "Often" and "rarely" are the answers, and it's a leading cause of a poor *perception* of quality by end users.

The product's release is a time for communication and education. Help users understand what is coming a long way off. Help them understand the benefits the application offers them (if there aren't any, revisit your requirements and ask why user-related challenges weren't addressed as requirements). Offer various forms of training to get users ready for the new application. Get them *excited* about it while at the same time resisting the urge to oversell it and create laughably high expectations that can never be met.

Setting Expectations

One of the biggest problems with a software release can be accurately setting user expectations—something Microsoft can attest to. Its Windows Vista release built up such high expectations and anticipation that literally no piece of software could have been successful. Technically, Vista *was* a success: it was relatively stable compared with its predecessors, offered a good suite of built-in features, and was highly anticipated by Windows users. But the expectations had been built so high over its 5-year development cycle that users were left wanting more and had a vague feeling that Vista was somehow a letdown.

That's the danger in getting overexcited: You *want* users to be excited about the benefits of a new software application but *only* if they can be *accurately* excited about what it really does for them. Overly high expectations result in, from the user perspective, poor quality; stable and rich as Vista was, it suffered from poor reviews in the media, bad user opinions (even from users who'd never seen it), and was generally perceived as a low-quality version of Windows—in large part due to expectations.

The release is also the time to start collecting requirements for the *next* release: What bugs need to be addressed? What business functionality—which will continue to evolve over time as the business changes and grows—needs to be added? New software often opens users' minds to new possibilities; begin capturing those to help drive a new set of requirements for an even better future version of the software. *Don't* take these new requirements as criticism; no matter how amazing a piece of software, it *always* generates new ideas for improvements, and that's a good thing—if nothing else, it keeps the development team employed!

Up Next: Costs, Barriers, and Benefits

In this chapter, we've taken a long, hard look at what comprises quality and come to the conclusion that the business perspective of quality is the best one to take: It reflects the realities of software development but does so from a viewpoint that is driven by the business' actual needs and requirements. Quality, in other words, should be viewed primarily in terms of its ability to provide value to the business. We've looked at how the various stages of the software development process can contribute quality (and, therefore, value). What we haven't examined are the costs of achieving quality, the costs of *not* having quality, and the barriers to (and benefits of) application quality. That's the focus on the next chapter: What quality will cost, and what will stand in your way of having it.

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.