Realtime publishers

The Definitive Guide™ To

Quality Application Delivery

sponsored by

MICRO FOCUS®
Leading the Evolution™

Don Jones

Copyright Statement

Chapter 5: Requirements—Quality from the Beginning

If you've read the previous four chapters in this guide, you've likely gotten the message that software quality is driven first and foremost by the business-level requirements you set at the beginning of the application development process. In this chapter, we'll look at requirements in great detail, with a goal of defining and describing exactly what it is that makes a good set of requirements—and how those requirements drive application quality.

In this book's first chapter, we agreed that "bug free" is one of the first and most common criteria most people cite when asked what they consider to be a "quality application." Defect-free software is certainly desirable, but it's not something you can actually specify in a set of requirements (of course, you *can* write that in your requirements—but it won't particularly help). For that reason, "bug free" is something we will *not* be discussing in this chapter. Instead, we're going to look at the business- and user-level requirements that lead to a quality application. Don't take the omission of bugs as a negative, however; we will definitely be looking at ways to improve application quality with respect to defects—but will be doing so in later chapters because decreasing the bug count is more a function of software design, development, and testing, rather than in requirements definition.

Requirements ≠ Evil

Defining application requirements is so important that the software development industry has cooked up an acronym for the final result: An *ARD* or *Application Requirements Document*. Whenever a new application project starts, many organizations embark on an epic process called *requirements gathering*, where the company attempts to discover, define, and document everything that the final application must be and do. In fact, sometimes the process goes a bit too far, taking vast amounts of time to generate unwieldy, unhelpful documents. Some businesses even get a bad taste in their collective mouths about application requirements, believing that the entire "gathering" phase is ultimately a waste of time that does nothing to improve the application.

Certainly, trying to comprehend the entire scope of a massive application, and clearly communicate its various requirements, can be an immense undertaking with plenty of room for things to go wrong. Many software development teams, in fact, now approach requirements from a much more piecemeal perspective, defining requirements for smaller, more self-contained chunks of functionality and letting those chunks accumulate into a completed application. This is essentially software modularization taken up to the business level, and it's a good approach for breaking down a complex application into more manageable units.

One of the problems with developing good requirements is that it's a soft process—it's easy for people within the organization to be confused about what they're supposed to do in order to contribute requirements, and easy for people to lose track of requirements in the press of day-to-day challenges. That's why tools exist to help make requirements documentation more streamlined and formal. These tools can help enforce a workflow so that requirements suggestions can be submitted, reviewers have a queue of suggestions to review and work with (and reminders to do so), and so forth. These tools can also help requirements documents from becoming overloaded by allowing screen shots, workflow diagrams, and other means of communication to be more easily incorporated into the document. Tools can also make it easier for teams or groups to review and discuss requirements, capture feedback, and incorporate changes.

Time Is an Enemy

Let's be brutally frank for a moment and discuss one of the top reasons that requirements—and thus entire software projects—go wrong: Time.

Software development schedules are nearly always determined by some factor *other than* how long the software will actually take to develop: competitive concerns, pressing business needs, economic factors, and so forth. Given a specific timeline—which is often set arbitrarily, at least from a software perspective—and a finite number of resources, you're only going to be able to accomplish a certain amount of work. Requirements are often seen as not contributing directly to software being released on time—that is, requirements are perceived as overhead not as "real" work. Thus, requirements are often the first thing to suffer, and the period allowed to gather requirements is often truncated or eliminated. That's a pity, because doing so invariably dooms the entire project to mediocrity at best.

Changing this attitude can't be done with tools, training, or even, in some cases, simple logic. In order for companies to achieve better software quality, they need to recognize any cultural or political shortcomings they may have, such as the tendency to set arbitrary software delivery deadlines and to shortchange critical aspects such as requirements gathering. After acknowledging these shortcomings, organizations need to work to change them, getting buy-in from the very top levels of management to ensure that higher-level pressures won't force a return to "business as usual."

No amount of software tools or other solutions can help an organization that is fundamentally unwilling to change. Be prepared to recognize that there are two reasons organizations don't produce quality software: One reason is simply a lack of proper tools and training, and that can be fixed; the other reason is a cultural or political environment that doesn't engender quality. Such an environment *can* be changed, but it often requires more work—and until it *is* changed, you can't begin to properly address tools and training.

Clearly Defining Business Needs—In Terms of Quality

Far too often, "requirements documents" going by names like *Software Requirements Specifications, Software Functional Specifications,* or *System Specifications* focus too much on application features. Writing about application features seems entirely straightforward: After all, you're describing what you want the application to do. But here's why functional requirements, by themselves, are insufficient:

They don't tell anyone why the business needs the functionality.

If it was possible to print that phrase in blinking red text, it would help communicate the severity of the situation. Even a well-written specification, which would contain non-functionality requirements related to performance and maintenance, perhaps, is ultimately incomplete if it doesn't also tie everything back to business needs.

Why, Why, Why?

There's no better way to explain why this is such a massive problem than an example, so read this excerpt from a "functional specification" document:

> The application must provide the ability for information from the database to be archived into a flat file. An administrator will specify the date range for data to be archived; the application must de-normalize all related data so that the archive data is internally complete and all references are included. The archival process must take no more than 10 minutes to complete, and must produce a file in the Microsoft Access database format.

That may seem like a really clear, precise set of functional requirements, and someone might be justifiably proud of turning such a document over to a software designer, and then on to programmers and testers. But that excerpt doesn't explain why, from a business perspective, any of those things are necessary. That means designers, developers, and testers are going to be left making a lot of assumptions—which will ultimately affect the quality of the application.

To continue with the example, suppose that a few things go wrong during the development process:

- Developers discover that, with a fully populated database, they're unable to meet the 10-minute requirement. When they attempt to make an archive from the database, after populating the database with expected production-level test data, exports take 30 minutes or more.
- Testers discover that they need to have Microsoft Access installed on their computers in order for the Microsoft Access export format to work properly. They know that not every administrator who will be making exports has Access installed, so they send a recommendation to the developers that a more universal format—one which is Access-compatible—be considered.

These are problems, and they need to be solved. It's not uncommon for something specified in a requirements document to be unachievable; quality software comes when the development team can make a smart decision about changes and compromises. Keep in mind that the original requirements authors might not be available any longer, and so the development team needs to make their decision based entirely on the intelligence included in the requirements document itself.

In our example, the developers make some assumptions and some important decisions:

- Archive exports will probably only be made once a month or even once a quarter, and the developers believe that an automated process, which can be run overnight, can be used to create the export files. Because a live administrator won't be involved, the time requirement isn't as critical.
- Developers agree that calling for an Access license is too strong a requirement, so they modify the application to export a bunch of comma-separated value (CSV) files. They know these can be imported into Access, and they write a standalone utility that can use the CSV files to construct an Access database if necessary. This meets the requirement of having an Access-format database but offers an option for users who don't have Access installed.

In making these assumptions, our developers have unknowingly compromised the quality of the application. Here's why:

- The archive export will actually be done several times each day because the archived data will be sent to a partner company. Because the export is done so frequently, it involves significantly less data than the developers realized—in fact, it would take less than 5 minutes in most cases. Also, the export will always be done by a live administrator because the exact export criteria will change based on business conditions that weren't known to the developers.
- The partner company will receive the exported data via an automated upload process that can accept only Access-formatted files. The company has already negotiated a volume license agreement for Microsoft Access so that everyone who needs it will have it.

The problem is that the original requirements were entirely *functional* in nature, with no consideration for the *business*. Developers didn't understand, from the requirements, what the application was being used for, and so they made incorrect assumptions both during development and testing. As a result, they're poised to deliver a low-quality application. They've spent extra time and money doing so, and will spend even more time and money correcting the situation. You could argue that they simply should have read and obeyed the requirements—but they're human beings, and humans will always attempt to understand the entire scope of something and to make smart, reasonable decisions. In this case, the humans just didn't have all the data.

A better set of requirements would also address the *business* needs:

> The application will need to export archive data, which will be sent to partner companies. This export will be done several times each day, and the export criteria will change continuously. A live person will do the export, and the export should use as little of their time as possible—the target is less than 10 minutes. The file will be sent to an automated import system that can read only Microsoft Access files. Each export must be complete so that no external references are needed to understand and use the data.

This statement isn't any longer than the previous one, but it describes more clearly how and why the application will be used. It still leaves room for assumptions to be made—in fact, it's impossible to write a document that doesn't leave room for assumptions. However, with the why more clearly stated, those assumptions can be more informed and will result in better decisions.

Further, this more clearly states the business' needs versus what the business perhaps thinks it wants. A designer may look at this revised requirement and realize that an off-the-shelf tool can do the job rather than having this functionality built as part of the application itself. Whenever possible, in fact, requirements should avoid making technical references, suggestions, or specifications, instead focusing on the business need. This gives a designer, and the development team, the freedom to select the most efficient technical means of meeting the business' needs.

Make It About the Needs

Ideally, a requirements document should start by simply listing all the benefits that the business expects to derive from the project. Do you expect to save money? How? Why? Do you expect to see productivity improvements? Why? How? Forget about the computers, at this point, and just explain why the business is undertaking the application development. In fact, to avoid confusion, you might not even refer to this document as a "requirements document" or "functional specification" or anything else. Call this the "business expectations document." From there, you can start developing a more detailed functional specification, making sure that everything specified in that document matches up and supports the business expectations.

What to Look for in a Good Requirements Document

A finished requirements document should achieve four major components:

- As stated, it should explain what the business expects to gain from the application—at a fairly detailed level. This shouldn't be a broad "mission statement" but rather an in-depth look at what the business wants from this application, along with the reasons why.
- It should give end users—that is, anyone who will use and rely upon the application—a feeling that the development team will understand what is needed. The development team doesn't work in the line of business every day, and so they don't know the little ins and outs of doing business to which the application must adhere. The requirements document is a way to transfer that day-to-day business knowledge to the development team. Again, though, focus just as much on the why as the what: Explaining that you need a particular report to run in 10 minutes doesn't help the development team understand why that time factor is critical—and not understanding will result in bad assumptions.
- A requirements document helps to organize the business' thoughts on the project and helps decompose a large problem into manageable units. By focusing on things such as end-user input, reporting, maintenance, and other areas individually, more thought goes into what the business needs to achieve from each. A good requirements document will be organized by these broad functional areas, which also help communicate the ways in which the application will be used.
- The requirements document should serve as a checklist for developers and testers. Essentially, the business should be able to look at the requirements document and the final application, go down the requirements list, and see which ones the application meets. If the application meets them all, and is relatively free of bugs, by definition it is a "quality application." By the same token, developers and testers should be able to continually refer to the requirements document; if what they're producing results in items from the document being "checked off," they know they're on the right track.

What's Too Much?

The further a requirements document strays into technical specifics, the less useful it becomes. This document isn't a means of telling technical people how to do *their* jobs; it's a means of telling technical people how to do the job of *someone else* in the business. Technology shouldn't enter into it.

That said, a requirements document can contain technical-looking information if the information derives from the day-to-day business needs that the application must address. For example, a requirements document doesn't need to specify a database structure—but it *should* provide a great deal of detail about what pieces of data the application must collect, store, and maintain. Don't assume a developer knows what information is contained in a "customer order," but don't feel the need to provide an entity-relationship diagram (ERD), either.

A Requirements Document Template

It can sometimes be helpful to start with a template outline, just to make sure your requirements document is covering everything it should. Here's a sample:

- Scope—In this section, identify the application, provide a brief description of what it does and where it fits into the business, and describe any security restrictions around the requirements document itself.
- General Characteristics—This is usually the biggest section, and lists all the business-level characteristics of the application. Be sure to include the *why* information whenever applicable! Broadly, you may include any of the following sub-sections:
    - Audiences—A general description of who will use the application.
    - External interfaces—What external software will interface with the application? How? Why?
    - Data—Describe the data the application will collect, store, and maintain—including any requirements (such as retention periods) around that.
    - Safety—Will the application have any safety impact, such as controlling external machinery?
    - Security and privacy—What concerns are there about the data the application stores? What requirements must be met, and why? It's important to note requirements that are internal versus those imposed by external legislation of industry rules.
    - Computing base—Where will the application operate? Will it run on desktop computers, mobile devices, or a combination of those? What kind of processing power will the application need to run on?
    - Constraints—Is there anything technical that's off-limits? Must any specific development or design technologies be used? For example, has the company decided to standardize on C# code or a particular Web browser?
    - Personnel—Are there restrictions on personnel training? Will the application need to perform in the same way as some existing application in order to minimize training? What personnel are available for maintenance?
    - Logistics—Are there are requirements around application deployment and ongoing updates?
    - Performance—Are there specific performance requirements that must be met?

  - Maintenance—How will the application be maintained over the long term? How will its data be maintained?
  - Monitoring—How will the application be monitored for health and performance?
  - Impact—Are there any restrictions on how the application impacts the current environment? Consider network performance, storage resources, and other possible pitfalls, to communicate the fact that the application will not run in a vacuum but must rather coexist in a production environment.
  - Precedence—Identify requirements that are "must haves" versus those that are "nice to have."
- Use Cases—This should specify the common job tasks that the application will enable. How will users interact with the application? What workflow will the application enable? Flowcharts and mocked-up screenshots are ideal at this point because they help communicate to the development team how the end users *want* to use the application.

There are a number of characteristics you should look for in a requirements document:

- **Complete.** The document should precisely define all the conditions that identify an application that will be perceived by the business as "quality."
- **Consistent.** The document must not contain any internal contradictions that are not resolved through some sort of priority ranking.
- **Real-World.** The document defines the system's capabilities and performance in a real-world environment, and identifies how the application and the real world interact.
- **Organized.** The document should be organized so that it is easily updated and modified. Future versions of the application may start with this same document, meaning the new version's features should focus on modifications to this document.
- **Prioritized.** Individual requirements must be ranked in some fashion, using whatever scale the business prefers.
- **Testable.** Every requirement in the document must be testable, which means every requirement must be definitive and unambiguous. You must be able to look at a requirement, then look at the application, and determine whether that application definitively meets that requirement. For requirements that are not simple "present or not" (for example, the presence of a feature), indicate precise measurements in units of time, data storage, or whatever ("the task must require less than 5 minutes to complete by existing staff members who have had 4 hours of training and 1 week of practice").

- **Traceable.** Every requirement has a "why." That might be an internal business requirement, an external legal requirement, or whatever—but the document must explain the why for each requirement.
- **Unambiguous.** Every requirement must include only one possible interpretation. Don't use ambiguous words such as "fast" or "small;" use unambiguous measurements and descriptions.

To help achieve some of these goals, there are clear, less-ambiguous words that a requirements document can use:

- Shall (or shall not)
- Must (or must not)
- Is required to
- Responsible for
- Will

> **Note**
> Some frameworks and formal templates have very specific definitions for these and other words—be sure to observe those definitions when they're in use.

Avoid "soft" words such as *should*, *may, might,* and so forth. In the remaining sections of this chapter, we'll look at specific examples of how to write requirements that have these desired characteristics.

Specifying Quality Indicators in the Requirements

I've been very specific with the title of this section: Specifying *quality indicators.* Try not to approach your requirements document as a giant punch list of business desires. Instead, think of it as a list of criteria that will be used to determine whether the application is of high quality. If the application is and does everything in the requirements document, your business will love it. In this section, we'll look at specific examples of quality indicators—and how to word them—that should go into your requirements.

Safety Requirements

Safety isn't something we often think of when it comes to software applications. After all, how likely is a piece of software to threaten life and limb? Well, that all depends on what the software is, where it's going to be used, and how it's going to be used. For example, consider an application that will be used to dispatch delivery drivers to their delivery locations. You might very reasonably want to consider the safety implications of drivers behind the wheel of a delivery truck using a software application:

> The application will not be usable while the delivery vehicle is in motion. The application may give verbal prompts for driving directions, route changes, and other updates, but will display a primarily static screen and will not permit user interaction.

This is a well-written quality indicator:

- It uses unambiguous language: "The application will not be usable…" and "…will display a primarily static screen…"
- It offers options to the design and development team: "…may give verbal prompts." This tells the design team that prompting is allowed but that the precedence is to keep the screen from becoming a distraction.
- It doesn't seek to make any technology decisions. There's no indication of how the application will know that the vehicle is in motion or not; it merely sets out an implicit requirement that the application be able to sense motion. The design team gets to determine how to implement this requirement.
- It serves as a clear checklist: Anyone can examine the application when it's done and determine whether this requirement has been met.

This is a good example of a "pass/fail" quality indicator. In other words, the final application either meets this requirement or it doesn't; there's no a lot of room for interpretation. Whenever a requirement can be worded in this kind of unambiguous, "pass/fail" manner, do so: You'll have fewer assumptions and interpretations going on during the design and development stage.

Sometimes, it can be difficult to phrase requirements in such a clear, unambiguous manner. For example, consider this quality indicator for a manufacturing robot in an automobile factory:

> The robot will immediately cease operation whenever a human worker's safety is threatened. When such a threat becomes apparent, the robot will cease all motion and activity in less than 2 seconds, moving less than half an inch before stopping completely.

This *seems* like a clear requirement. It's certainly full of exact details, specifying that the robot stop moving within 2 seconds and move less than half an inch while ceasing operation. But how does the robot determine that the "…worker's safety is threatened?" What constitutes a threat? Should the robot cease moving if the worker trips and falls outside the factory or if there is a terrorist attack? There's not enough information for the design team to work with, and it would be difficult to examine the final application and determine whether this requirement had been met. A better requirement would remove any need for the designer to make assumptions:

> The robot will immediately cease operation whenever a human worker enters the robot's range of motion, as determined by sensors or other devices. When such an incursion happens, the robot will cease all motion and activity in less than 2 seconds, moving less than half an inch before stopping completely.

In this case, the requirement is clearer about the type of "threat" that the robot must be able to sense and respond to, and indicates that additional devices of same nature—to be determined by the designer—will be used to sense a threat condition.

Lots of people think that requirements documents can become overly detailed and cumbersome, but when it comes to physical safety, you can't *ever* be too detailed. As another example, consider the software that will be used to control a retail anti-theft device. The device itself uses magnetic waves to detect anti-theft tags embedded in merchandise, and the software is responsible for controlling the magnetic emissions. Do you think the following requirement is "overly detailed?"

> The application must modulate the magnetic field to avoid interference with any electrical devices, such as pacemakers.

"Well, of course it should do that," you might think. "There's no need to spell it out." But there *is.* Software designers and developers aren't necessarily aware that magnetic fields can damage or disrupt pacemakers—you can't assume that this is common knowledge and that the designers will automatically take it into account. Further, you can't assume that your testers will know about this, and think to test for it. By stating this simple quality indicator, you've set in motion a chain of design decisions that will include research on the proper modulation techniques, test cases that might well involve the use of actual pacemakers, and so forth. By *not* stating the requirement, your first test might well have been when a pacemaker-equipped customer walked through the device in a retail store.

Security and Privacy Requirements

Modern applications always have to consider security and privacy, and it's not enough to assume that designers and developers will consider this automatically. In fact, two decades of software development history strongly suggest that security and privacy are often the *last* thing a designer or developer considers.

If your security or privacy requirements come from a previously documented, or external, set of requirements, it is sufficient to re-state those requirements as quality indicators:

> The application will conform to all relevant sections of the Sarbanes-Oxley Act with regard to the security of financial information.
>
> The application will confirm to corporate security standard 72.4 with regard to the storage of customer personal information.

Those external standards become incorporated, by reference, into the requirements document—and it's a good idea to make sure they're included physically or electronically with the document, as well.

> **Tip**
> One reason that requirements-management tools exist is to help manage the "document attachments" that you will inevitably have as you flesh out your requirements.

Security and privacy nearly always have potential negative business impacts, and software designers and developers are frequently unaware of them. For example, a movie rental business that accidentally discloses a customer's rental history might find itself on the wrong end of a class-action lawsuit—a possibility that will probably not even occur to the developers who are creating the software that stores the rental history. Because of the severe potential business impact, it's worth your time to spell out, as quality indicators, the exact security your application requires.

> The application will not disclose rental history information to anyone outside the corporate headquarters. Within the corporate headquarters, only specifically authorized individuals (not members of an entire group or department) will have access to rental history for any given customer. All access to individual customer history will be logged in a tamperproof database, and those access records will be maintained for 7 years. The application will permanently delete and securely erase all individual rental history older than 12 months. This requirement applies only to rental history associated with specific customers; aggregate rental history not associated with an individual person will be treated as "marketing data" and handled s described elsewhere in these requirements.

Doesn't get any clearer than that, right? Unfortunately, this excellent start on a quality indicator still leaves a lot of room for imagination. Here's a somewhat more thorough version, with an addition in boldface:

> The application will not disclose rental history information to anyone outside the corporate headquarters, **and the application will be designed to prevent disclosure of this information to anyone who bypasses the application (for example, direct access to database files)**. Within the corporate headquarters, only specifically authorized individuals (not members of an entire group or department) will have access to rental history for any given customer. All access to individual customer history will be logged in a tamperproof database, and those access records will be maintained for 7 years. The application will permanently delete and securely erase all individual rental history older than 12 months. This requirement applies only to rental history associated with specific customers; aggregate rental history not associated with an individual person will be treated as "marketing data" and handled s described elsewhere in these requirements.

Yes, that's a nitpick, but history and experience suggests that software designers and developers aren't traditionally very imaginative when it comes to security. For example, an application that stores data in clear-text files can be easily bypassed, resulting in the exact kind of privacy breach you were trying to prevent—but you never *said* that you wanted the files encrypted. With the revised statement, you're making it clear that not only must the application itself have appropriate security controls but also the design of the application must prevent the application's own controls from being bypassed. You're not specifying technologies such as encryption; that's a decision for the designer. You're merely specifying the business requirements, and in this case, those requirements have been stated in a way that drives design, development, and testing.

This is, of course, where you also have to decide how much quality you can afford. The previously stated requirement is going to create a significant amount of design and development work, but it's going to create a ton of testing work. Your testing team is essentially going to have to "play hacker" and attempt to crack the application and its database to determine whether this quality requirement is being met. The result could be an expensive testing process for something that isn't a likely business risk. To help save money, you might make one further change to your quality indicator (again, changes are in boldface):

> The application will not disclose rental history information to anyone outside the corporate headquarters, and the application will be designed to prevent disclosure of this information to anyone who bypasses the application (for example, direct access to database files). **If the application encrypts this data using the strongest encryption available and the designer demonstrates that encryption keys are secure, that will be taken as positive evidence of this requirement being met.** Within the corporate headquarters, only specifically authorized individuals (not members of an entire group or department) will have access to rental history for any given customer. All access to individual customer history will be logged in a tamperproof database, and those access records will be maintained for 7 years. The application will permanently delete and securely erase all individual rental history older than 12 months. This requirement applies only to rental history associated with specific customers; aggregate rental history not associated with an individual person will be treated as "marketing data" and handled s described elsewhere in these requirements.

Here, the requirement has dipped its toes into the technical waters and specified a means by which the application can demonstrate its compliance with this requirement *without* extensive "black hat" testing. You won't actually be *testing* for compliance, so you're taking a bit more of a business risk, but in this case, you may judge that the cost of a full test outweighs the potential risk. This is another great example of the business being firmly in the driver's seat, deciding not only what the application must achieve but also on any compromises.

Computing Base Requirements

It's easy to think of "computing base" entirely in terms of performance, as in:

> The application must run on single-core processors of more than 1.8GHz, on computers having 2GB or more of RAM, and must move between input screens with a delay of less than 2 seconds at all times.

And although that is an excellent sort of requirement—it gives your testing team a very clear set of criteria and gives developers a very clear target to hit—that is not the beginning and end of establishing your computing base requirements. You might also have requirements designed to ensure the long-term maintainability of the application's code:

> The application will be developed using Microsoft .NET Framework tools and technologies.

You might set other requirements that force the application to take advantage of existing resources rather than bringing in new ones (and the accompanying support and licensing costs):

> The application will use Microsoft SQL Server 2008 for all back-end database storage.

Other requirements might be designed to ensure consistency with other systems and applications:

> The application will use Microsoft SQL Server Reporting Services for all reporting.

Finally, other requirements might establish a functional baseline for the application:

> The application will run on Microsoft Windows XP Service Pack 2 or later, including Windows 7. It will run without requiring administrative privileges of the end user.

These are obviously all technical requirements, an unavoidable aspect of anything related to your computing base. In essence, these represent constraints on the designer and developers, telling them that certain technology decisions have been made in advance, and that they will be expected to comply with and support those decisions.

Personnel Requirements

There are two types of personnel requirements, and both—as with computing base requirements—essentially act as constraints on the design and development team. The first type specifies what is generally referred to as *accessibility* requirements:

> The application will be compliant with section 508 of the disabilities act.

Don't add these types of requirements unless you *must* have them because these requirements can add a significant price to design, development, and testing. For example, if you know the application will not need full 508 compliance, you may want to specify a subset:

> The application will be accessible to people with any form of color blindness.

The level of accessibility required of an application depends entirely upon business factors: Large defense contractors, for example, are often required to maintain full 508 compliance, as are most government agencies. Smaller private companies may not be under the same requirements. Because the differences are legally complex, you shouldn't make your designers, developers, and testers decide what to do—give them clear instruction in the form of well-worded quality indicators.

The second form of personnel requirement is designed to increase usability. These can range from specifications for user interface design:

> The application user interface will utilize Microsoft Office 2007 conventions.

To requests that the application mirror the workflow of an existing system:

> Whenever possible, end-user workflow will be identical to the workflow in the existing LCARS 7.2 software. Deviations from that workflow are acceptable for features that do not exist in v7.2, and for instances where a deliberate workflow change is mandated by accessibility requirements or by changing user workflow (as indicated in the included use cases).

This is a short, sweet, and delightful quality indicator. Look at all the great things its doing:

- It establishes a baseline for workflow using an existing software application. This significantly reduces the need to collect use cases and other workflow-definition documents because much of the workflow is already defined.
- It leaves an "out" for instances where the existing workflow simply can't be matched to the new application's functionality.
- It establishes precedence between this requirement and another requirement for accessibility, making it clear that accessibility carries a higher priority.
- It also acknowledges that some existing workflows may be outdated or no longer acceptable, and offers a means by which new workflows will be specified. However, it's clear to any reader that workflows that are not explicitly overridden by a new use case or accessibility concerns will be maintained as-is.

Is there a need for this type of personnel requirement? Absolutely. What you're doing here is minimizing end-user training, which can be a significant cost when it comes time to deploy the application. You're also allowing users to continue using familiar workflows—something that users will perceive as a "better" application. You're offering the opportunity to change workflows that users don't currently like, giving you a "win" that will help users look forward to the new application. Finally, you're meeting what may be a legal obligation by specifying accessibility as a priority.

Other Functional Requirements

Although I've lumped these under the category of "other," that doesn't make them less important. In fact, these requirements are simply the ones that most people think of most easily—the requirements that spell out your application's capabilities and features. Don't limit yourself to this list, but definitely start with these:

- Input—How will data get into the system? Data entry? Imports? Web services?
- Output—How will data leave the system? Exports? Web services? Reports? On-screen displays?
- Reports—Consider anything that might be printed, including management reports, end-user reports, financial reports, tax reports, shipping labels, price tags, and so forth.

- Configuration—What operating parameters will the application have that might change over time? You might want to specify that these be easily changeable without needing a new version release of the software.
- Processing—What business rules and algorithms will the software use? Whenever possible, provide these business rules as flowcharts or narratives so that designers and developers can see exactly how the application should "think."

Those last two are the ones that are most frequently overlooked or underdeveloped. For example, an online retailer that prints thousands of shipping labels a day might specify something like this:

> The application will have the ability to print shipping labels for a variety of carriers and service levels. Each carrier+service level combination may have a different label. Label layouts will be designed as easily replaceable templates so that when the carrier changes their labeling requirements the application can be adjusted without reprogramming. When an order is placed, the application will user the order's shipping method and carrier to select the appropriate template and produce the appropriate label.

This requirement does some excellent things for the application:

- It clearly states that label layouts will need to change over time.
- It explains why labels should be made from templates—because there's a desire to change templates without reprogramming. The word "template" is a bit vague, but combined with the business reason, there's sufficient context for a designer to make the right decisions.
- It communicates the fact that a given carrier may require different labeling for different service levels—something a developer may not be aware of.
- The implication is that it's the carriers, not the company, who change label layouts. A smart software designer might take this as a cue to investigate the carriers and see whether they offer their own templates, which can be consumed by the application. Doing so would make it even easier to adopt changes in the future.

When it comes to business logic, almost nothing beats a flowchart. Consider this requirement:

> When a customer places an item in their shopping cart, inventory for that item is checked and the item is not added if there is not sufficient available inventory. Instead, an "out of stock" notice is displayed. Adding the item to the cart does not debit inventory, so inventory must be checked again when the order is finalized, and any items that are out of inventory must be removed from the cart and the order finalized automatically.

That's not unambiguous, but it's not as clear as Figure 5.1.

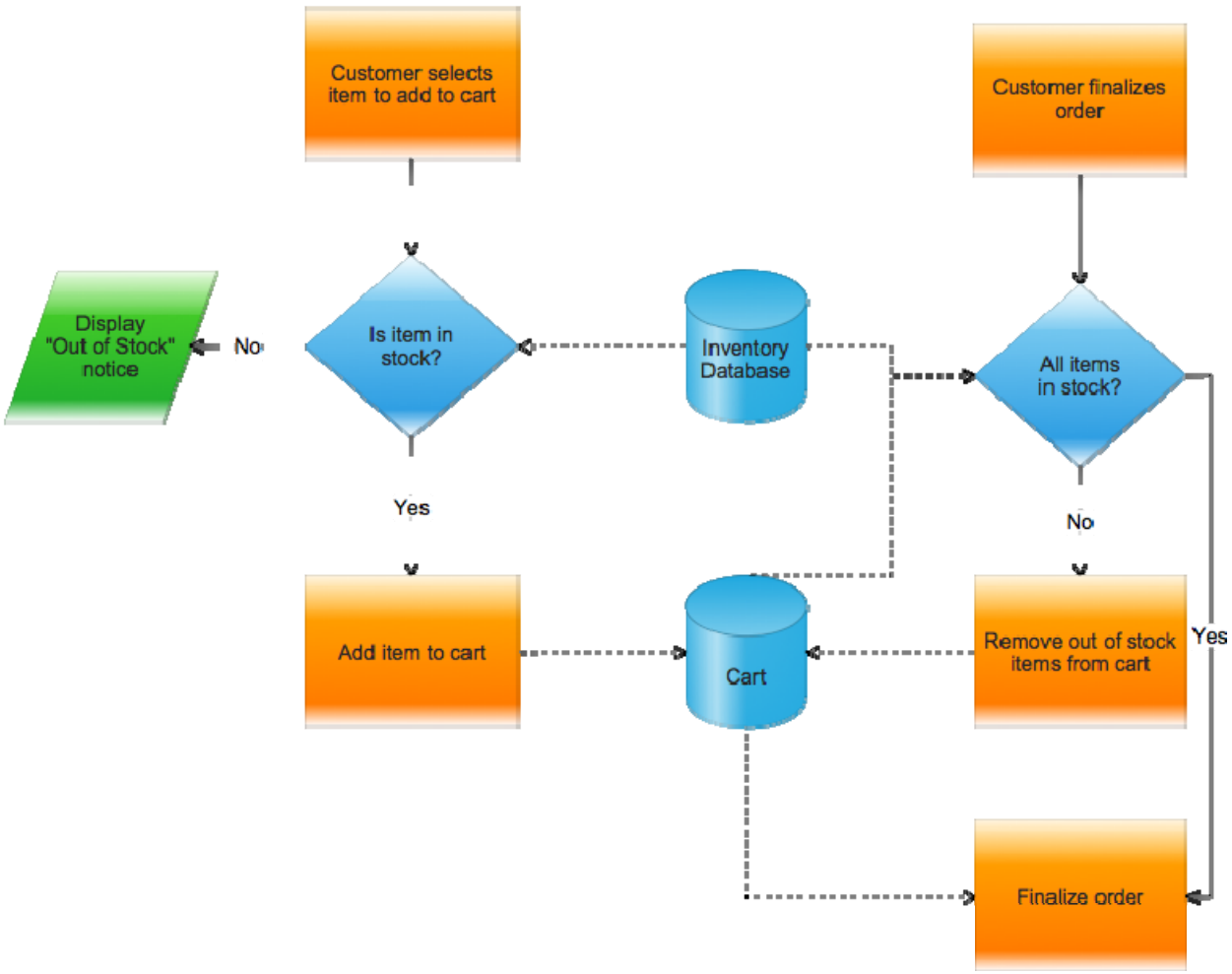

Figure 5.1: Business logic expressed in a flowchart.

One reason flowcharts work better for describing business logic is that a software designer or developer can more easily add notations to indicate which aspect of the software implements each rule. This makes debugging and troubleshooting much easier, and serves as a better checklist to make sure that everything was implemented as desired. Figure 5.2 shows an annotated version of this business logic, mapping the functionality to the actual implementation.

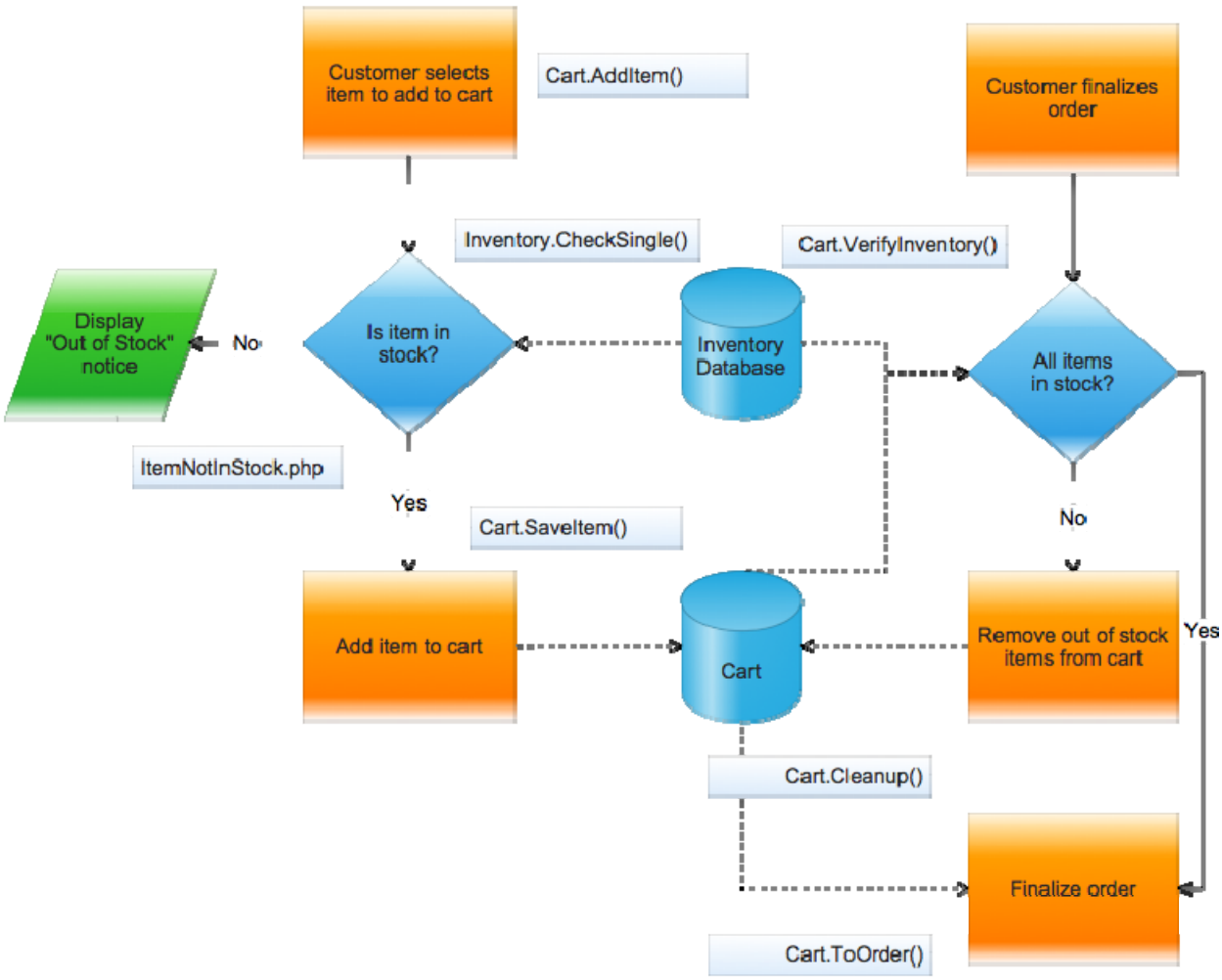

Figure 5.2: Annotated business process maps the process to its implementation.

Specifying Performance Metrics and Other Non-Functional Requirements

In addition to all the requirements related to the proper functioning of the software, you've also got to worry about non-functional requirements—that is, requirements that don't relate to features or capabilities but refer to *how* the software functions. Performance and monitoring are perhaps two of the most important, and often-overlooked, non-functional requirements.

Nobody wants a slow-running application, and next to "bugs" most users will cite "performance" as the biggest indicator of what they perceive as a low-quality application. Yet many businesses never even consider performance when they're defining the requirements for a new application! When considering performance requirements, there are a number of considerations to think about and specify:

- Where will the application run? On your network, across the Internet, or on a partner or customer network? Can you specify the level of traffic that already exists to give designers an idea of how much they've got left to work with in terms of bandwidth?
- Will the application be allowed to use all available resources on the network, a database server, a Web server, or elsewhere? Or will the application need to limit the resources it uses?
- How long should certain key tasks take to complete? This should include not only obvious tasks performed by end users but also report generation, maintenance tasks, and so forth.
- How many people will be using this application? Where will it be deployed (internally, on the Internet, etc.)?
- How much workload will those users generate? Will every user be hammering at it nonstop or is it only an occasional-use application?

It's easy to see that last bullet as being important, but in fact it's useless without the first two. Developers who are testing their code in a development environment can come up with whatever performance they need; it's when the application hits the real world that performance expectations start to get hurt. By describing as best as possible the real-world conditions that the application will run under, you can drive design, development, and testing to include those real-world parameters. Requirements can be written like this example:

> The existing production environment in which the application will run is a 100MB Ethernet network with 1000MB connectivity between servers and switches. The network is 62% utilized on average, and this application must add no more than 20% additional utilization. This application is replacing an application that is responsible for approximately 15% of the current utilization.

Note

Remember, quality costs. If you can't already describe, in meaningful technical terms, what your existing environment looks like and how the application will have to behave in it, then expect a designer to have to perform some testing and analysis to determine that information. That will take time and cost money.

Another thing you need to specify is how the application's health and performance will be monitored over the long term:

> The application must be instrumented for monitoring via standard Windows performance counters and via Windows Management Instrumentation. The application must provide performance metrics for key internal tasks (identified under "key tasks" elsewhere in these requirements). The application does not need to instrument back-end database performance if the database software already provides those metrics.

This is important because eventually, given enough time and growth, *every application will become slow.* You need to have a way of monitoring the application's health on a regular basis, establishing performance trends, and spotting the slowdown before it becomes a problem. If you're already using a companywide management solution, you might want to specify that the application integrate with that solution—although, again, that may drive up development and testing costs:

> The application must expose performance information to Microsoft System Center Operations Manager, and the development team must create a Management Pack that translates performance data into health indicators.

Specifying Maintenance Requirements

Maintenance is another frequently-overlooked set of requirements that you need to consider. These are functional requirements but they're more back-end; some of the things you need to think about include:

- How will the application be deployed? Do you have an existing software deployment mechanism that should be utilized?
- How will updates be deployed?
- How will data be archived to keep the production database running smoothly? How will archived data be accessed when needed?
- How will users be granted access to the application? How will this access be changed or revoked as time passes?

Each application will have its own maintenance needs, but don't expect developers to think of these and come up with solutions. Consider your administrators—those who will be maintaining the application—as another kind of end user. *They* know what they like and dislike about application maintenance, so use their expertise to help drive the maintenance requirements for your new application.

Specifying Use Cases

People get nervous about writing use cases because they offer a lot of opportunity to accidentally omit important information. So consider *not* writing use cases. Instead, get actual users from each audience to write a *user story:*

> As a salesperson, I want to be able to enter customer orders more quickly so that I can have a conversation with the customer without having to constantly say, "the computer is slow right now, it'll be up in a second." When I enter customer orders, I always start with the customer's name, if they don't know their ID number. Not many know their ID number. From there, I'd like to be able to enter everything on one screen—address changes, whatever items they are ordering—all of it in one place. It takes too long to switch between screens.

Although this is a short story, it tells a lot about how this particular user will interact with the application. It also tells a lot about their frustrations with the current application they're using, and what needs to be done to help solve those frustrations and increase productivity and customer satisfaction.

We've learned that customer records are retrieved by customer name or ID number. We've also learned that the current application requires a lot of switching between views in order to enter and edit data, and that the current system is slow. It's probably a good idea to assume that the new system will become slow over time, so the user's suggestion of keeping all the information in a single view is a good one. It is easy to think, "Well, the new system will be faster, and the user's concern is clearly about speed, so we can discount the idea of keeping everything on one screen." Try to avoid that thinking if possible: The user's perception of quality is going to be directly related to how improved the new application is over the old one; switching between screens is perceived as a problem of the old application; by avoiding that workflow, you can generate an immediate perception of better quality.

Encourage multiple users from the same audience to submit user stories—that gives you a broader perspective, captures more information about how the day-to-day business works, and captures more information about weaknesses with the current system. Encourage users to submit sketches of data entry screens, reports, and other interfaces, if desired. In many cases, users can do a better job of laying out the information so that it makes sense to them than a designer or developer could. For example, Figure 5.3 illustrates a data-entry screen sketch submitted by a user. The layout of the data is a bit unusual, with address information off to the side rather than laid out as a mailing label with the customer's name—which is a more traditional user interface design approach.

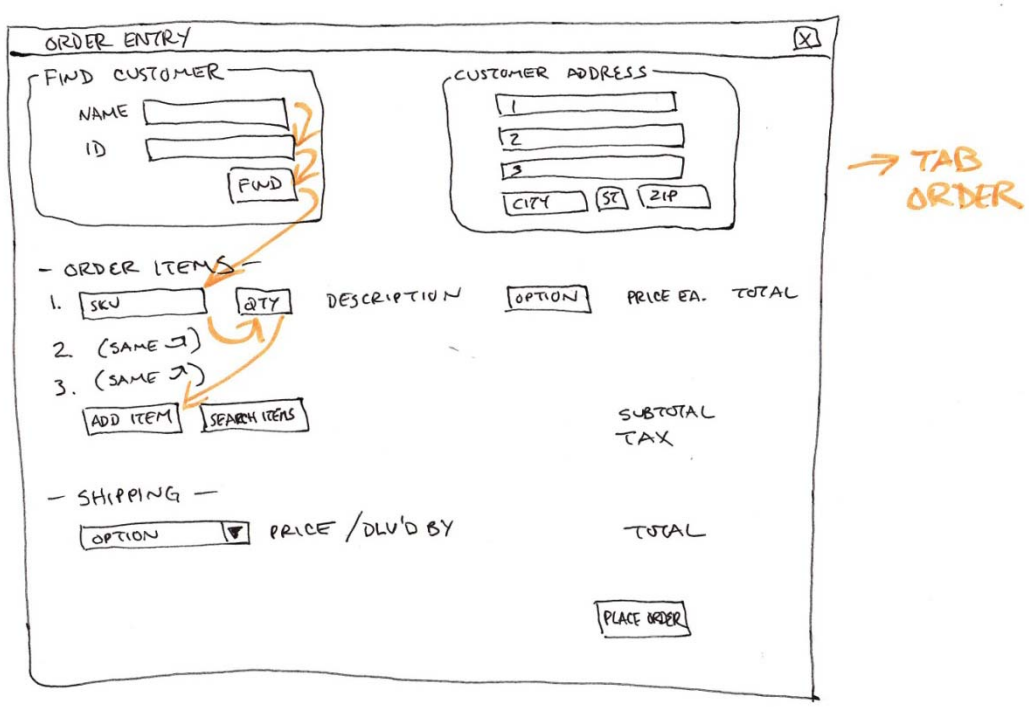

Figure 5.3: Using user drawings as part of a use case.

In this example, the narrative might reveal that address information is edited only rarely, and that once a salesperson retrieves a customer record, they want to start entering item numbers as quickly as possible. By using their keyboard to tab directly into item entry, thus bypassing the address entry, the salesperson feels they would be more productive—and while that might not be the final user interface design, it's extremely valuable intelligence that should definitely be taken into consideration during the design phase.

> **Tip**
>
> Having users include tab order hints on their drawings can be tremendously useful in documenting the precise workflow that users rely on to complete job tasks.

It's important to remember that user stories—and use cases—aren't intended to be requirements in and of themselves. Instead, they're intended to capture information about how the application will be used, what benefits each audience hopes to realize, and in many cases, what doesn't work about the existing system. This information is incorporated into the requirements document because it can help drive both design and development, and can provide valuable insight into what each audience will accept in terms of overall application quality.

Management in the Driver's Seat

The idea with a solid requirements document is to take incorrect assumptions away from software designers and developers, and to clearly communicate the *business* expectations related to a software development project. Doing so puts management in the driver's seat, ensuring that everyone associated with the project understands what the business drivers are, how the business expects the software to work, and what value the business expects the software to deliver. In other words, the requirements document explains *why* this software development project exists in the first place, and provides all the *correct* assumptions that the design and development team needs to make good decisions and produce a high-quality piece of software.

> **Note**
>
> Much has been written and said about the difficulty in outsourcing software development projects. You'll find that the ability to produce a thorough, business-oriented requirements document can go a long way toward making outsourcing more successful. Although outsourcing obviously requires ongoing management throughout the project's life cycle, starting with a clear, business-level requirements document helps ensure that the outsourced designers and developers clearly understand what you expect from the software and helps managers on your end ensure that your ultimate business needs are being met.

Coming Up Next: Designing for Quality

Your requirements are complete, and it's time to start making some technical decisions that will ultimately be used to implement your requirements. In other words, you're ready for the design phase. Although many organizations miss out on quality by not having a great requirements document, the design phase also leaves plenty of room for quality to be permanently lost. In the next chapter, we'll look at how the design process can be managed to add quality, rather than missing out on 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.