Part I: Software Requirements – What, Why, and Who

Chapter List

Chapter 1: The Essential Software Requirement

Overview

"Hello, Phil? This is Maria in Human Resources. We're having a problem with the employee system you programmed for us. An employee just changed her name to Sparkle Starlight, and we can't get the system to accept the name change. Can you help?"

"She married some guy named Starlight?"

"No, she didn't get married, just changed her name," Maria replied. "That's the problem. It looks like we can change a name only if someone's marital status changes."

"Well, yeah, I never thought someone might just change her name. I don't remember you telling me about this possibility when we talked about the system. That's why you can get to the Change Name dialog box only from the Change Marital Status dialog box," Phil said.

"I assumed you knew that people could legally change their name anytime they like," responded Maria. "We have to straighten this out by Friday or Sparkle won't be able to cash her paycheck. Can you fix the bug by then?"

"It's not a bug!" Phil retorted. "I never knew you needed this capability. I'm busy on the new performance evaluation system. I think I have some other change requests for the employee system here, too." [sound of rustling paper] "Yeah, here's another one. I can probably fix it by the end of the month, but not within a week. Sorry about that. Next time, tell me these things earlier and please write them down."

"What am I supposed to tell Sparkle?" demanded Maria. "She's really going to be ticked if she can't cash her check."

"Hey, Maria, it's not my fault," Phil protested. "If you'd told me in the first place that you had to be able to change someone's name at any time, this wouldn't have happened. You can't blame me for not reading your mind."

Angry and resigned, Maria snapped, "Yeah, well, this is the kind of thing that makes me hate computer systems. Call me as soon as you get it fixed, will you?"

If you've ever been on the customer side of a conversation like this, you know how frustrating it is to use a software product[1] that doesn't let you perform an essential task. You'd also rather not be at the mercy of a developer who *might* get to your critical change request eventually. Developers know how frustrating it is to learn of functionality that the user expects only after they've implemented the system. It's also annoying to have your current project interrupted by a request to modify a system that does precisely what you were told it should do in the first place.

Many software problems arise from shortcomings in the ways that people gather, document, agree on, and modify the product's requirements. As with Phil and Maria, the problem areas might include informal information gathering, implied functionality, erroneous or uncommunicated assumptions, inadequately defined requirements, and a casual change process.

Most people wouldn't ask a construction contractor to build a custom $300,000 house without extensively discussing their needs and desires and refining the details progressively. Homebuyers understand that making changes carries a price tag; they don't like it, but they understand it. However, people blithely gloss over the corresponding issues when it comes to software development. Errors made during the requirements stage account for 40 to 60 percent of all defects found in a software project (Davis 1993; Leffingwell 1997). The two most frequently reported problems in a large survey of the European software industry concerned specifying and managing customer requirements (ESPITI 1995). Nonetheless, many organizations still practice ineffective methods for these essential project activities. The typical outcome is an expectation gap, a difference between what developers think they are supposed to build and what customers really need.

Nowhere more than in the requirements process do the interests of all the stakeholders in a software or system project intersect. These stakeholders include

- Customers who fund a project or acquire a product to satisfy their organization's business objectives.
- Users who interact directly or indirectly with the product (a subclass of customers).
- Requirements analysts who write the requirements and communicate them to the development community.
- Developers who design, implement, and maintain the product.
- Testers who determine whether the product behaves as intended.
- Documentation writers who produce user manuals, training materials, and help systems.
- Project managers who plan the project and guide the development team to a successful delivery.
- Legal staff who ensure that the product complies with all pertinent laws and regulations.
- Manufacturing people who must build the products that contain software.
- Sales, marketing, field support, help desk, and other people who will have to work with the product and its customers.

Handled well, this intersection can lead to exciting products, delighted customers, and fulfilled developers. Handled poorly, it's the source of misunderstanding, frustration, and friction that

undermine the product's quality and business value. Because requirements are the foundation for both the software development and the project management activities, all stakeholders must be committed to following an effective requirements process.

But developing and managing requirements is hard! There are no simple shortcuts or magic solutions. However, so many organizations struggle with the same problems that we can look for techniques in common that apply to many different situations. This book describes dozens of such techniques. They're presented as though you were building a brand-new system, but most of the techniques also apply to maintenance projects and to selecting commercial off-the-shelf package solutions. (See Chapter 16, "Special Requirements Development Challenges.") Nor do these requirements engineering techniques apply only to projects that follow a sequential waterfall development life cycle. Even project teams that build products incrementally need to understand what goes into each increment.

This chapter will help you to

- Understand some key terms used in software requirements engineering.
- Distinguish requirements development from requirements management.
- Be alert to some requirements-related problems that can arise.
- Learn several characteristics of excellent requirements.

Taking Your Requirements Pulse

For a quick check of the current requirements engineering practices in your organization, ask yourself how many of the following conditions apply to your most recent project. If you check more than three or four boxes, this book is for you.

- The project's vision and scope are never clearly defined.
- Customers are too busy to spend time working with analysts or developers on the requirements.
- User surrogates, such as product managers, development managers, user managers, or marketers, claim to speak for the users, but they don't accurately represent user needs.
- Requirements exist in the heads of "the experts" in your organization and are never written down.
- Customers claim that all requirements are critical, so they don't prioritize them.
- Developers encounter ambiguities and missing information when coding, so they have to guess.
- Communications between developers and customers focus on user interface displays and not on what the users need to do with the software.
- Your customers sign off on the requirements and then change them continuously.
- The project scope increases when you accept requirements changes, but the schedule slips because no additional resources are provided and no functionality is removed.
- Requested requirements changes get lost, and you and your customers don't know the status of all change requests.

- Customers request certain functionality and developers build it, but no one ever uses it.
- The specification is satisfied, but the customer is not.

[1]I use the terms *product*, system, and *application* interchangeably in this book. The concepts and practices I discuss apply to any kind of software or software-containing item that you build.

Software Requirements Defined

One problem with the software industry is the lack of common definitions for terms we use to describe aspects of our work. Different observers might describe the same statement as being a user requirement, software requirement, functional requirement, system requirement, technical requirement, business requirement, or product requirement. A customer's definition of *requirements* might sound like a high-level product concept to the developer. The developer's notion of requirements might sound like detailed user interface design to the user. This diversity of definitions leads to confusing and frustrating communication problems.

True Stories A key concept is that the requirements must be documented. I was on a project once that had experienced a rotating cast of developers. The primary customer was sick to tears of having each new requirements analyst come along and say, "We have to talk about your requirements." The customer's reaction was, "I already gave your predecessors the requirements. Now build me a system!" In reality, no one had ever documented the requirements, so every new analyst had to start from scratch. To proclaim that you have the requirements is delusional if all you really have is a pile of e-mail and voice-mail messages, sticky notes, meeting minutes, and vaguely recollected hallway conversations.

Some Interpretations of Requirement

Consultant Brian Lawrence suggests that a *requirement* is "anything that drives design choices" (Lawrence 1997). Many kinds of information fit in this category. The *IEEE Standard Glossary of Software Engineering Terminology* (1990) defines a requirement as

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

This definition encompasses both the user's view of the requirements (the external behavior of the system) and the developer's view (some under-the-hood characteristics). The term *user* should be generalized to *stakeholder* because not all stakeholders are users. I think of a requirement as a property that a product must have to provide value to a stakeholder. The following definition acknowledges the diversity of requirements types (Sommerville and Sawyer 1997):

> *Requirements are…a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.*

Clearly, there's no universal definition of what a requirement is. To facilitate communication, we need to agree on a consistent set of adjectives to modify the overloaded term *requirement*, and we need to appreciate the value of recording these requirements in a shareable form.

Trap Don't assume that all your project stakeholders share a common notion of what requirements are. Establish definitions up front so that you're all talking about the same things.

Levels of Requirements

This section presents definitions that I will use for some terms commonly encountered in the requirements engineering domain. Software requirements include three distinct levels—business requirements, user requirements, and functional requirements. In addition, every system has an assortment of nonfunctional requirements. The model in Figure 1-1 illustrates a way to think about these diverse types of requirements. As with all models, it is not all-inclusive, but it provides a helpful organizing scheme. The ovals represent types of requirements information and the rectangles indicate containers (documents, diagrams, or databases) in which to store that information.

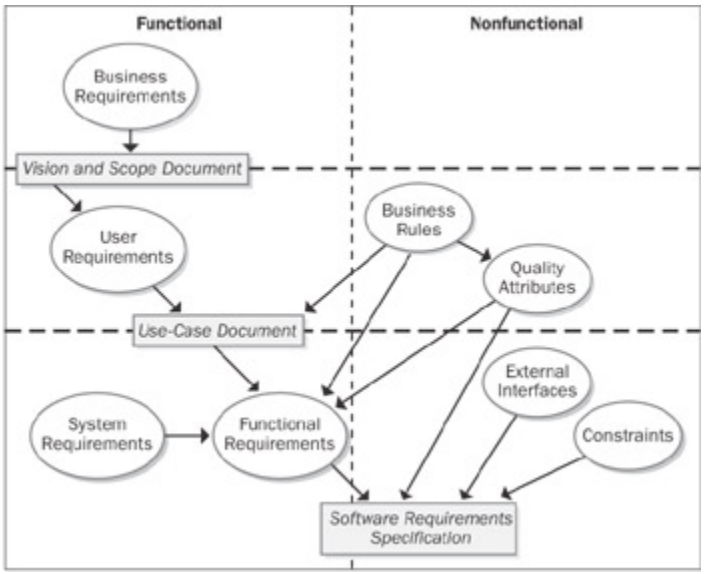

Figure 1-1: Relationship of several types of requirements information.

More Info Chapter 7, "Hearing the Voice of the Customer," contains many examples of these different types of requirements.

Business requirements represent high-level objectives of the organization or customer who requests the system. Business requirements typically come from the funding sponsor for a project, the acquiring customer, the manager of the actual users, the marketing department, or a product visionary. Business requirements describe why the organization is implementing the system—the objectives the organization hopes to achieve. I like to record the business requirements in a *vision and scope document*, sometimes called a *project charter* or a *market requirements document*. Creating such a document is the subject of Chapter 5, "Establishing the Product Vision and Project Scope." Defining the project scope is the first step in controlling the common problem of scope creep.

User requirements describe user goals or tasks that the users must be able to perform with the product. Valuable ways to represent user requirements include use cases, scenario descriptions, and event-response tables. User requirements therefore describe what the user will be able to do with the system. An example of a use case is "Make a Reservation" using an airline, a rental car, or a hotel Web site.

More Info Chapter 8, "Understanding User Requirements," addresses user requirements.

Functional requirements specify the software functionality that the developers must build into the

product to enable users to accomplish their tasks, thereby satisfying the business requirements. Sometimes called *behavioral requirements*, these are the traditional "shall" statements: "The system shall e-mail a reservation confirmation to the user." As Chapter 10 ("Documenting the Requirements") illustrates, functional requirements describe what the developer needs to implement.

The term *system requirements* describes the top-level requirements for a product that contains multiple subsystems—that is, a *system* (IEEE 1998c). A system can be all software or it can include both software and hardware subsystems. People are a part of a system, too, so certain system functions might be allocated to human beings.

True Stories My team once wrote software to control some laboratory apparatus that automated the tedious addition of precise quantities of chemicals to an array of beakers. The requirements for the overall system led us to derive software functional requirements to send signals to the hardware to move the chemical-dispensing nozzles, read positioning sensors, and turn pumps on and off.

Business rules include corporate policies, government regulations, industry standards, accounting practices, and computational algorithms. As you'll see in Chapter 9, "Playing By the Rules," business rules are not themselves software requirements because they exist outside the boundaries of any specific software system. However, they often restrict who can perform certain use cases or they dictate that the system must contain functionality to comply with the pertinent rules. Sometimes business rules are the origin of specific quality attributes that are implemented in functionality. Therefore, you can trace the genesis of certain functional requirements back to a particular business rule.

Functional requirements are documented in a *software requirements specification* (SRS), which describes as fully as necessary the expected behavior of the software system. I'll refer to the SRS as a document, although it can be a database or spreadsheet that contains the requirements, information stored in a commercial requirements management tool—see Chapter 21, "Tools for Requirements Management"—or perhaps even a stack of index cards for a small project. The SRS is used in development, testing, quality assurance, project management, and related project functions.

In addition to the functional requirements, the SRS contains nonfunctional requirements. These include performance goals and descriptions of quality attributes. *Quality attributes* augment the description of the product's functionality by describing the product's characteristics in various dimensions that are important either to users or to developers. These characteristics include usability, portability, integrity, efficiency, and robustness. Other nonfunctional requirements describe external interfaces between the system and the outside world, and design and implementation constraints. *Constraints* impose restrictions on the choices available to the developer for design and construction of the product.

People often talk about product features. A *feature* is a set of logically related functional requirements that provides a capability to the user and enables the satisfaction of a business objective. In the commercial software arena, a feature is a group of requirements recognizable to a stakeholder that aids in making a purchase decision—a bullet item in the product description. A customer's list of desired product features is not equivalent to a description of the user's task-related needs. Web browser favorites or bookmarks, spell check, macro recording, automobile power windows, online update of tax code changes, telephone speed-dialing, and automatic virus signature updating are examples of product features. A feature can encompass multiple use cases, and each use case requires that multiple functional requirements be implemented to allow the user to perform the task.

To get a better grasp on some of the different kinds of requirements I've been discussing, consider a word processing program. A business requirement might read, "The product will allow users to correct spelling errors in a document efficiently." The product's box cover announces that a spell checker is included as a feature that satisfies this business requirement. Corresponding user requirements might include tasks—use cases—such as "Find spelling errors" and "Add word to global

dictionary." The spell checker has many individual functional requirements, which deal with operations such as finding and highlighting a misspelled word, displaying a dialog box with suggested replacements, and globally replacing misspelled words with corrected words. The quality attribute called *usability* would specify just what is meant by the word "efficiently" in the business requirement.

Managers or marketing define the business requirements for software that will help their company operate more efficiently (for information systems) or compete successfully in the marketplace (for commercial products). All user requirements must align with the business requirements. The user requirements permit the analyst to derive the bits of functionality that will let the users perform their tasks with the product. Developers use the functional and nonfunctional requirements to design solutions that implement the necessary functionality and achieve the specified quality and performance objectives, within the limits that the constraints impose.

Although the model in Figure 1-1 shows a top-down flow of requirements, you should expect cycles and iteration between the business, user, and functional requirements. Whenever someone proposes a new feature, use case, or functional requirement, the analyst must ask, "Is this in scope?" If the answer is "yes," the requirement belongs in the specification. If the answer is "no," it does not. If the answer is "no, but it ought to be," the business requirements owner or the funding sponsor must decide whether to increase the project scope to accommodate the new requirement. This is a business decision that has implications for the project's schedule and budget.

What Requirements Are Not

Requirements specifications do not include design or implementation details (other than known constraints), project planning information, or testing information (Leffingwell and Widrig 2000). Separate such items from the requirements so that the requirements activities can focus on understanding what the team intends to build. Projects typically have other kinds of requirements, including development environment requirements, schedule or budget limitations, the need for a tutorial to help new users get up to speed, or requirements for releasing a product and moving it into the support environment. These are *project* requirements but not *product* requirements; they don't fall within the scope of this book.

PREVIOUS NEXT
PREVIOUS NEXT

Requirements Development and Management

Confusion about requirements terminology extends even to what to call the whole discipline. Some authors call the entire domain *requirements engineering* (Sommerville and Kotonya 1998); others refer to it as *requirements management* (Leffingwell and Widrig 2000). I find it useful to split the domain of software requirements engineering into *requirements development* (addressed in Part II of this book) and *requirements management* (addressed in Part III), as shown in Figure 1-2.

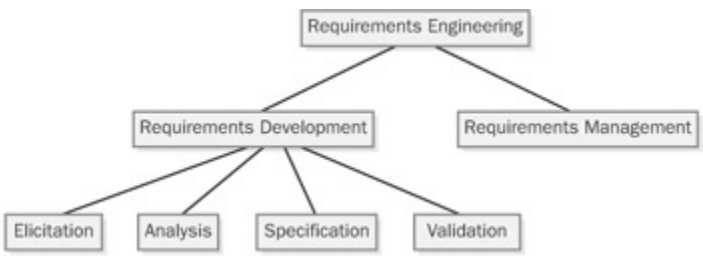

Figure 1-2: Subcomponents of the requirements engineering domain.

Requirements Development

We can further subdivide requirements development into *elicitation*, *analysis*, *specification*, and *validation* (Abran and Moore 2001). These subdisciplines encompass all the activities involved with gathering, evaluating, and documenting the requirements for a software or software-containing product, including the following:

- Identifying the product's expected user classes
- Eliciting needs from individuals who represent each user class
- Understanding user tasks and goals and the business objectives with which those tasks align
- Analyzing the information received from users to distinguish their task goals from functional requirements, nonfunctional requirements, business rules, suggested solutions, and extraneous information
- Allocating portions of the top-level requirements to software components defined in the system architecture
- Understanding the relative importance of quality attributes
- Negotiating implementation priorities
- Translating the collected user needs into written requirements specifications and models
- Reviewing the documented requirements to ensure a common understanding of the users' stated requirements and to correct any problems before the development group accepts them

Iteration is a key to requirements development success. Plan for multiple cycles of exploring requirements, refining high-level requirements into details, and confirming correctness with users. This takes time and it can be frustrating, but it's an intrinsic aspect of dealing with the fuzzy uncertainty of defining a new software product.

Requirements Management

Requirements management entails "establishing and maintaining an agreement with the customer on the requirements for the software project" (Paulk et al. 1995). That agreement is embodied in the written requirements specifications and the models. Customer acceptance is only half the equation needed for requirements approval. The developers also must accept the specifications and agree to build them into a product. Requirements management activities include the following:

- Defining the requirements baseline (a snapshot in time representing the currently agreed-upon body of requirements for a specific release)
- Reviewing proposed requirements changes and evaluating the likely impact of each change before approving it
- Incorporating approved requirements changes into the project in a controlled way
- Keeping project plans current with the requirements
- Negotiating new commitments based on the estimated impact of requirements changes

- Tracing individual requirements to their corresponding designs, source code, and test cases
- Tracking requirements status and change activity throughout the project

Figure 1-3 provides another view of the distinction between requirements development and requirements management. This book describes several dozen specific practices for performing requirements elicitation, analysis, specification, validation, and management.

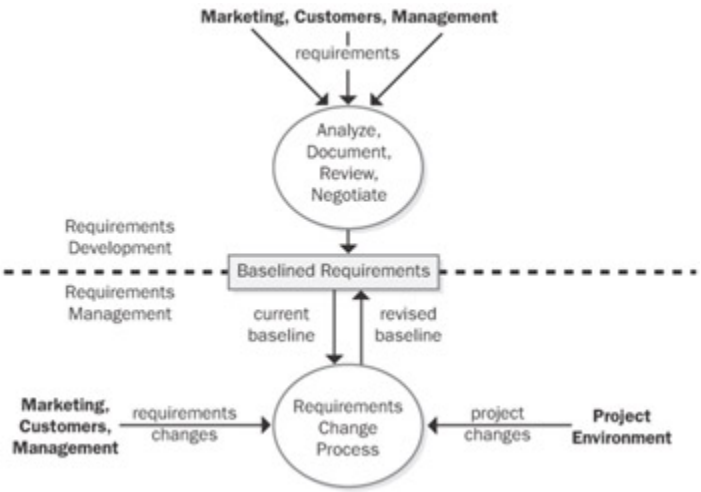

Figure 1-3: The boundary between requirements development and requirements management.

Every Project Has Requirements

Frederick Brooks eloquently stated the critical role of requirements to a software project in his classic 1987 essay, "No Silver Bullet: Essence and Accidents of Software Engineering":

> *The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.*

Every software application or software-containing system has users who rely on it to enhance their lives. The time spent understanding user needs is a high-leverage investment in project success. If developers don't have written requirements that the customers agree to, how can they satisfy those customers? Even the requirements for software that isn't intended for commercial use must be well understood. Examples include software libraries, components, and tools created for internal use by a development group.

Often, it's impossible to fully specify the requirements before commencing construction. In those cases, take an iterative and incremental approach to implement one portion of the requirements at a time, obtaining customer feedback before moving on to the next cycle. This isn't an excuse to write code before contemplating requirements for that next increment, though. Iterating on code is more expensive than iterating on concepts.

People sometimes balk at spending the time that it takes to write software requirements, but writing the requirements isn't the hard part. The hard part is *discovering* the requirements. Writing requirements is primarily a matter of clarifying, elaborating, and transcribing. A solid understanding

of a product's requirements ensures that your team works on the right problem and devises the best solution to that problem. Requirements let you prioritize the work and estimate the effort and resources needed for the project. Without knowing what the requirements are, you can't tell when the project is done, determine whether it has met its goals, or make trade-off decisions when scope reduction is necessary.

No Assumed Requirements

True Stories I recently encountered a development group that used a homegrown software engineering tool that included a source code editor. Unfortunately, no one had specified that the tool should permit users to print the code, although the users undoubtedly assumed that it would. Developers had to hand-transcribe the source statements to hold a code review.

If you don't write down even the implicit and assumed requirements, don't be surprised if the software doesn't meet user expectations. Periodically ask, "What are we assuming?" to try to surface those hidden thoughts. If you come across an assumption during requirements discussions, write it down and confirm its accuracy. If you're developing a replacement system, review the previous system features to determine whether they're required in the replacement rather than assuming that they are or are not.

When Bad Requirements Happen to Nice People

The major consequence of requirements problems is rework—doing over something that you thought was already done. Rework can consume 30 to 50 percent of your total development cost (Boehm and Papaccio 1988), and requirements errors account for 70 to 85 percent of the rework cost (Leffingwell 1997). As illustrated in Figure 1-4, it costs far more to correct a defect that's found late in the project than to fix it shortly after its creation (Grady 1999). Preventing requirements errors and catching them early therefore has a huge leveraging effect on reducing rework. Imagine how different your life would be if you could cut the rework effort in half! You could build products faster, build more and better products in the same amount of time, and perhaps even go home occasionally.

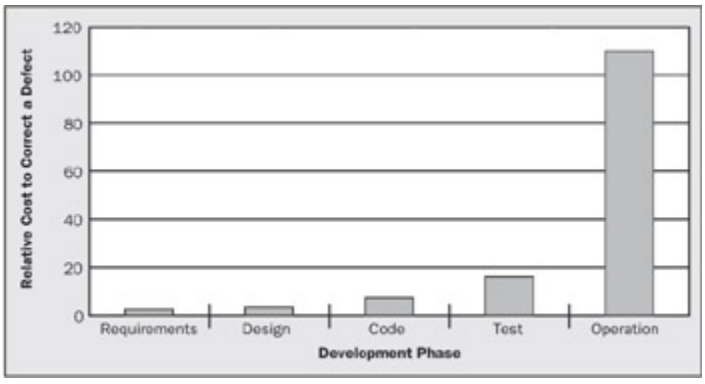

Figure 1-4: Relative cost to correct a requirement defect depending on when it is discovered.

Shortcomings in requirements practices pose many risks to project success, where *success* means delivering a product that satisfies the user's functional and quality expectations at agreed-on cost and schedule. Chapter 23, "Software Requirements and Risk Management," describes how to manage such risks to prevent them from derailing your project. Some of the most common requirements risks are described in the following sections.

Insufficient User Involvement

Customers often don't understand why it is so essential to work hard on gathering requirements and assuring their quality. Developers might not emphasize user involvement, either because working with users isn't as much fun as writing code or because they think they already know what the users need. In some cases, it's difficult to gain access to people who will actually use the product, and user surrogates don't always understand what users really need. Insufficient user involvement leads to late-breaking requirements that delay project completion. There's no substitute for having the development team work directly with actual users throughout the project, as described in Chapter 6.

Creeping User Requirements

As requirements evolve and grow during development, projects often exceed their planned schedules and budgets. Such plans aren't always based on realistic understandings of the size and complexity of the requirements; constant requirements modifications make the problem worse. The problem lies partly in the users' frequent requests for changes in the requirements and partly in the way that developers respond to these requests.

To manage scope creep, begin with a clear statement of the project's business objectives, strategic vision, scope, limitations, success criteria, and expected product usage. Evaluate all proposed new features or requirements changes against this reference framework. An effective change process that includes impact analysis will help the stakeholders make informed business decisions about which changes to accept and the associated costs in time, resources, or feature trade-offs. Change is often critical to success, but change always has a price.

As changes propagate through the product being developed, its architecture can slowly crumble. Code patches make programs harder to understand and maintain. Added code can cause modules to violate the solid design principles of strong cohesion and loose coupling (McConnell 1993). To minimize this type of quality degradation, flow requirements changes through the architecture and design rather than implementing them directly into code.

Ambiguous Requirements

Ambiguity is the great bugaboo of requirements specifications (Lawrence 1996). One symptom of ambiguity is that a reader can interpret a requirement statement in several ways. Another sign is that multiple readers of a requirement arrive at different understandings of what it means. Chapter 10 lists many words and phrases that contribute to ambiguity by placing the burden of interpretation on the reader. Ambiguity also results from imprecise and insufficiently detailed requirements that force developers to fill in the blanks.

Ambiguity leads to different expectations on the part of various stakeholders. Some of them are then surprised with whatever is delivered. Ambiguous requirements result in wasted time when developers implement a solution for the wrong problem. Testers who expect the product to behave differently from what the developers intended waste time resolving the differences.

One way to ferret out ambiguity is to have people who represent different perspectives formally inspect the requirements. (See Chapter 15, "Validating the Requirements," for more on this subject.) Simply passing around the requirements document for comments isn't sufficient to reveal ambiguities. If different reviewers interpret a requirement in different ways but it makes sense to each of them, the ambiguity won't surface until late in the project, when correcting it can be expensive. Writing test cases against the requirements and building prototypes are other ways to discover ambiguities. Gause and Weinberg (1989) describe additional ambiguity-detection methods.

Gold Plating

Gold plating takes place when a developer adds functionality that wasn't in the requirements specification but that the developer believes "the users are just going to love." Often users don't care about this excess functionality, and the time spent implementing it is wasted. Rather than simply inserting new features, developers and analysts should present the customers with creative ideas and alternatives for their consideration. Developers should strive for leanness and simplicity, not for going beyond what the customer requests without customer approval.

Customers sometimes request features or elaborate user interfaces that look cool but add little value to the product. Everything you build costs time and money, so you need to maximize the delivered value. To reduce the threat of gold plating, trace each bit of functionality back to its origin so that you know why it's included. The use-case approach for eliciting requirements helps to focus requirements elicitation on the functionality that lets users perform their business tasks.

Minimal Specification

Sometimes marketing staff or managers are tempted to create a limited specification, perhaps just a product concept sketched on a napkin. They expect the developers to flesh out the spec while the project progresses. This might work for a tightly integrated team that's building a small system, on exploratory or research projects, or when the requirements truly are flexible (McConnell 1996). In most cases, though, it frustrates the developers (who might be operating under incorrect assumptions and with limited direction) and disappoints the customers (who don't get the product they envisioned).

Overlooked User Classes

Most products have several groups of users who might use different subsets of features, have different frequencies of use, or have varying experience levels. If you don't identify the important user classes for your product early on, some user needs won't be met. After identifying all user classes, make sure that each has a voice, as discussed in Chapter 6, "Finding the Voice of the Customer."

Inaccurate Planning

"Here's my idea for a new product; when will you be done?" Don't answer this question until you know more about the problem being discussed. Vague, poorly understood requirements lead to overly optimistic estimates, which come back to haunt us when the inevitable overruns occur. An estimator's off-the-cuff guess sounds a lot like a commitment to the listener. The top contributors to poor software cost estimation are frequent requirements changes, missing requirements, insufficient communication with users, poor specification of requirements, and insufficient requirements analysis (Davis 1995). When you present an estimate, provide either a range (best case, most likely, worst case) or a confidence level ("I'm 90 percent sure I can have that done within three months").

PREVIOUS NEXT
PREVIOUS NEXT

Benefits from a High-Quality Requirements Process

Organizations that implement effective requirements engineering processes can reap multiple benefits. A great reward comes from reducing unnecessary rework during the late development stages and throughout the lengthy maintenance period. The high leveraging effect of quality requirements isn't obvious, and many people mistakenly believe that time spent discussing requirements simply delays delivery by the same duration. A holistic cost-of-quality perspective reveals the value of emphasizing early-stage quality practices (Wiegers 1996a).

Sound requirements processes emphasize a collaborative approach to product development that

involves multiple stakeholders in a partnership throughout the project. Collecting requirements enables the development team to better understand its user community or market, a critical success factor for any project. It's far cheaper to reach this understanding before you build the product than after your customers receive it.

Engaging users in the requirements-gathering process generates enthusiasm for the product and builds customer loyalty. By emphasizing user tasks instead of superficially attractive features, the team can avoid writing code that won't ever be used. Customer involvement reduces the expectation gap between what the user needs and what the developer delivers. You're going to get the customer feedback eventually. Try to get it early rather than late, perhaps with the help of prototypes that stimulate user feedback. Requirements development takes time, but it takes less time than correcting a lot of problems in beta testing or after release.

There are additional benefits, too. Explicitly allocating selected system requirements to various software, hardware, and human subsystems emphasizes a systems approach to product engineering. An effective change-control process will minimize the adverse impact of requirements changes. Documented, unambiguous requirements greatly facilitate system testing, which in turn increases your chances of delivering high-quality products that satisfy all stakeholders.

No one can promise a specific return on investment from an improved requirements process. You can go through an analytical thought process to imagine how better requirements could help you, though. First, consider the cost of investing in a better process. This includes the cost of assessing your current practices, developing new procedures and document templates, training the team, buying books and tools, and perhaps using outside consultants. Your greatest investment is the time your teams spend gathering, documenting, reviewing, and managing their requirements. Next, think about the possible benefits you might enjoy and how much time or money they could save you. It's impossible to quantify all the following benefits, but they are real:

- Fewer requirements defects
- Reduced development rework
- Fewer unnecessary features
- Lower enhancement costs
- Faster development
- Fewer miscommunications
- Reduced scope creep
- Reduced project chaos
- More accurate system-testing estimates
- Higher customer and team member satisfaction

Team LiB PREVIOUS NEXT
Team LiB PREVIOUS NEXT

Characteristics of Excellent Requirements

How can you distinguish good requirements specifications from those with problems? Several characteristics that individual requirement statements should exhibit are discussed in this section, followed by desirable characteristics of the SRS as a whole (Davis 1993; IEEE 1998b). The best way to tell whether your requirements have these desired attributes is to have several project stakeholders carefully review the SRS. Different stakeholders will spot different kinds of problems. For example, analysts and developers can't accurately judge completeness or correctness, whereas users can't assess technical feasibility.

Requirement Statement Characteristics

In an ideal world, every individual user, business, and functional requirement would exhibit the qualities described in the following sections.

Complete

Each requirement must fully describe the functionality to be delivered. It must contain all the information necessary for the developer to design and implement that bit of functionality. If you know you're lacking certain information, use *TBD* (to be determined) as a standard flag to highlight these gaps. Resolve all TBDs in each portion of the requirements before you proceed with construction of that portion.

Correct

Each requirement must accurately describe the functionality to be built. The reference for correctness is the source of the requirement, such as an actual user or a high-level system requirement. A software requirement that conflicts with its parent system requirement is not correct. Only user representatives can determine the correctness of user requirements, which is why users or their close surrogates must review the requirements.

Feasible

It must be possible to implement each requirement within the known capabilities and limitations of the system and its operating environment. To avoid specifying unattainable requirements, have a developer work with marketing or the requirements analyst throughout the elicitation process. The developer can provide a reality check on what can and cannot be done technically and what can be done only at excessive cost. Incremental development approaches and proof-of-concept prototypes are ways to evaluate requirement feasibility.

Necessary

Each requirement should document a capability that the customers really need or one that's required for conformance to an external system requirement or a standard. Every requirement should originate from a source that has the authority to specify requirements. Trace each requirement back to specific voice-of-the-customer input, such as a use case, a business rule, or some other origin.

Prioritized

Assign an implementation priority to each functional requirement, feature, or use case to indicate how essential it is to a particular product release. If all the requirements are considered equally important, it's hard for the project manager to respond to budget cuts, schedule overruns, personnel losses, or new requirements added during development.

More Info Chapter 14, "Setting Requirement Priorities," discusses prioritization in further detail.

Unambiguous

All readers of a requirement statement should arrive at a single, consistent interpretation of it, but natural language is highly prone to ambiguity. Write requirements in simple, concise, straightforward language appropriate to the user domain. "Comprehensible" is a requirement quality goal related to "unambiguous": readers must be able to understand what each requirement is saying. Define all specialized terms and terms that might confuse readers in a glossary.

Verifiable

See whether you can devise a few tests or use other verification approaches, such as inspection or demonstration, to determine whether the product properly implements each requirement. If a requirement isn't verifiable, determining whether it was correctly implemented becomes a matter of opinion, not objective analysis. Requirements that are incomplete, inconsistent, infeasible, or ambiguous are also unverifiable (Drabick 1999).

Requirements Specification Characteristics

It's not enough to have excellent individual requirement statements. Sets of requirements that are collected into a specification ought to exhibit the characteristics described in the following sections.

Complete

No requirements or necessary information should be absent. Missing requirements are hard to spot because they aren't there! Chapter 7 suggests some ways to find missing requirements. Focusing on user tasks, rather than on system functions, can help you to prevent incompleteness.

Consistent

Consistent requirements don't conflict with other requirements of the same type or with higher-level business, system, or user requirements. Disagreements between requirements must be resolved before development can proceed. You might not know which single requirement (if any) is correct until you do some research. Recording the originator of each requirement lets you know who to talk to if you discover conflicts.

Modifiable

You must be able to revise the SRS when necessary and to maintain a history of changes made to each requirement. This dictates that each requirement be uniquely labeled and expressed separately from other requirements so that you can refer to it unambiguously. Each requirement should appear only once in the SRS. It's easy to generate inconsistencies by changing only one instance of a duplicated requirement. Consider cross-referencing subsequent instances back to the original statement instead of duplicating the requirement. A table of contents and an index will make the SRS easier to modify. Storing requirements in a database or a commercial requirements management tool makes them into reusable objects.

Traceable

A traceable requirement can be linked backward to its origin and forward to the design elements and source code that implement it and to the test cases that verify the implementation as correct. Traceable requirements are uniquely labeled with persistent identifiers. They are written in a structured, fine-grained way as opposed to crafting long narrative paragraphs. Avoid lumping multiple requirements together into a single statement; the different requirements might trace to different design and code elements.

More Info Chapter 10 discusses writing high-quality functional requirements, and Chapter 20, "Links in the Requirements Chain," addresses requirements tracing.

You'll never create an SRS in which *all* requirements demonstrate *all* these ideal attributes. However, if you keep these characteristics in mind while you write and review the requirements, you will produce better requirements documents and you will build better products.

Next Steps

- Write down requirements-related problems that you have encountered on your current or previous project. Identify each as a requirements development or requirements management problem. Identify the impact that each problem had on the project and its root causes.
- Facilitate a discussion with your team members and other stakeholders regarding requirements-related problems from your current or previous projects, the problems' impacts, and their root causes. Explain that the participants have to confront these difficult issues if they ever hope to master them. Are they ready to try?
- Arrange a one-day training class on software requirements for your entire project team. Include key customers, marketing staff, and managers, using whatever it takes to get them into the room. Training is an effective team-building activity. It gives project participants a common vocabulary and a shared understanding of effective techniques and behaviors so that they all can begin addressing their mutual challenges.

Chapter 2: Requirements from the Customer's Perspective

Overview

Gerhard, a senior manager at Contoso Pharmaceuticals, was meeting with Cynthia, the new manager of Contoso's information systems (IS) development group. "We need to build a chemical-tracking information system for Contoso," Gerhard began. "The system should let us keep track of all the chemical containers we already have in the stockroom and in individual laboratories. That way, maybe the chemists can get what they need from someone down the hall instead of buying a new container from a vendor. This should save us a lot of money. Also, the Health and Safety Department needs to generate some reports on chemical usage for the government. Can your group build this system in time for the first compliance audit five months from now?"

"I see why this project is important, Gerhard," said Cynthia. "But before I can commit to a schedule, we'll need to collect some requirements for your system."

Gerhard was confused. "What do you mean? I just told you the requirements."

"Actually, you described a concept and some business objectives for the project," Cynthia explained. "Those high-level business requirements don't give me enough detail to know what software to build or how long it might take. I'd like to have a requirements analyst work with some of the users to understand their needs for the system. Then we can figure out what functionality will meet both your

business objectives and the users' needs. You might not even need a new software system to meet your goal of saving money."

Gerhard hadn't encountered this reaction before from an IS person. "The chemists are busy people," he protested. "They don't have time to nail down every detail before you can start programming. Can't your people figure out what to build?"

Cynthia tried to explain her rationale for collecting requirements from the people who would use the new system. "If we just make our best guess at what the users need, we can't do a good job. We're software developers, not chemists. We don't really know what the chemists need to do with the chemical-tracking system. I've learned that if we don't take the time to understand the problem before we start writing code, nobody is happy with the results."

"We don't have time for all that," Gerhard insisted. "I gave you my requirements. Now just build the system, please. Keep me posted on your progress."

Conversations like this take place regularly in the software world. Customers who request a new information system often don't understand the importance of obtaining input from actual users of the proposed system in addition to other internal and external stakeholders. Marketing specialists with a great new product concept believe that they adequately represent the interests of prospective buyers. However, there's no substitute for gathering requirements directly from the people who will actually use the product. Some contemporary "agile" software development methodologies such as Extreme Programming recommend that a full-time, on-site customer work closely with the development team. As one book about Extreme Programming pointed out, "The project is *steered* to success by the customer and programmers working in concert" (Jeffries, Anderson, and Hendrickson 2001).

Part of the requirements problem results from confusion over the different levels of requirements: business, user, and functional. Gerhard stated some business requirements, benefits that he hopes Contoso will enjoy with the help of the new chemical-tracking system. However, Gerhard can't describe the user requirements because he is not an intended user of the system. Users, in turn, can describe the tasks they must be able to perform with the system, but they can't identify all the functional requirements that developers must implement to let them accomplish those tasks.

This chapter addresses the customer-development relationship that is so critical to software project success. I propose a Requirements Bill of Rights for Software Customers and a corresponding Requirements Bill of Responsibilities for Software Customers. These lists underscore the importance of customer—specifically user—involvement in requirements development.

Fear of Rejection

True Stories I recently heard a sad story when I visited a corporate information systems department. The developers had recently built a new system for use within the company. They had obtained negligible user input from the beginning. The day the developers proudly unveiled their new system, the users rejected it as completely unacceptable. This came as a shock because the developers had worked hard to satisfy what they perceived to be the users' needs. So what did they do? They fixed it. You always fix the system when you get the requirements wrong, and it always costs much more than if you had engaged user representatives from the outset.

The time the developers had to spend fixing the flawed information system wasn't planned, of course, so the next project that the team was scheduled to work on had to wait. This is a lose-lose-lose situation. The developers were embarrassed and frustrated, the customers were unhappy because their new system wasn't available for use when they expected it, and the company wasted a lot of money. Extensive and ongoing customer engagement from the start could have prevented this unfortunate, but not uncommon, project outcome.

Who Is the Customer?

In the broadest sense, a *customer* is an individual or organization who derives either direct or indirect benefit from a product. Software customers include those project stakeholders who request, pay for, select, specify, use, or receive the output generated by a software product. As we saw in Chapter 1, other project stakeholders include requirements analysts, developers, testers, documentation writers, project managers, support staff, legal staff, and marketing staff.

Gerhard, the manager we met earlier, represents the kind of customer who is paying for or sponsoring a software project. Customers at Gerhard's senior management level are responsible for defining the business requirements. They provide the high-level concept for the product and the business rationale for launching it. As discussed in Chapter 5, "Establishing the Product Vision and Product Scope," *business requirements* describe the business objectives that the customer, company, or other stakeholders want to achieve. Business requirements establish a guiding framework for the rest of the project. All other product features and requirements ought to align with satisfying the business requirements. However, business requirements don't provide sufficient detail to tell developers what to build.

The next level of requirements—user requirements—should come from people who will actually use the product, directly or indirectly. These users (often called *end users*) therefore constitute another kind of customer. Users can describe the tasks they need to perform with the product and the quality characteristics they expect the product to exhibit.

Customers who provide the business requirements sometimes purport to speak for the users, but they are usually too far removed from the actual work to provide accurate user requirements. For information systems, contract development, or custom application development, business requirements should come from the person with the money, whereas user requirements should come from people who will press the keys to use the product. The various stakeholders will need to check for alignment between the user and business requirements.

Unfortunately, both kinds of customers might not believe that they have the time to work with the requirements analysts who gather, analyze, and document the requirements. Sometimes customers expect the analysts or developers to figure out what users need without a lot of discussion. If only it were that easy. The days of sliding some vague requirements and a series of pizzas under the door to the programming department are long past.

The situation is somewhat different for commercial (shrink-wrapped) software development, in which the customer and the user often are the same person. Customer surrogates, such as the marketing department or product management, typically attempt to determine what customers would find appealing. Even for commercial software, though, you should engage end users in the process of developing user requirements, as Chapter 7 ("Hearing the Voice of the Customer") describes. If you don't do this, be prepared to read magazine reviews that describe shortcomings in your product that adequate user input could have helped you avoid.

Not surprisingly, conflicts can arise between business requirements and user requirements. Business requirements sometimes reflect organizational strategies or budgetary constraints that aren't visible to users. Users who are upset about having a new information system forced on them by management might not want to work with the software developers, viewing them as the harbingers of an undesired future. Clear communication about project objectives and constraints might defuse these tensions.

Perhaps not everyone will be happy with the realities, but at least they will have an opportunity to understand and buy into them. The analyst should work with the key user representatives and management sponsors to reconcile any conflicts.

The Customer-Development Partnership

Excellent software products are the result of a well-executed design based on excellent requirements. High-quality requirements result from effective communication and collaboration between developers and customers—a partnership. Too often, the relationship between development and customers becomes adversarial. Managers who override user-supplied requirements to suit their own agenda also can generate friction. No one benefits in these situations.

A collaborative effort can work only when all parties involved know what they need to be successful and when they understand and respect what their collaborators need to be successful. As project pressures rise, it's easy to forget that all stakeholders share a common objective: to build a successful software product that provides adequate business value and rewards to all stakeholders.

The Requirements Bill of Rights for Software Customers—see Table 2-1—lists 10 expectations that customers can legitimately hold regarding their interactions with analysts and developers during the project's requirements engineering activities. Each of these rights implies a corresponding responsibility on the part of the software developers or analysts. Conversely, the Requirements Bill of Responsibilities for Software Customers—see Table 2-2—lists 10 responsibilities that the customer has to the analyst and developer during the requirements process. You might prefer to view these as a developer's bill of rights.

Table 2-1: Requirements Bill of Rights for Software Customers

You have the right to
1. Expect analysts to speak your language.
2. Expect analysts to learn about your business and your objectives for the system.
3. Expect analysts to structure the information you present during requirements elicitation into a written software requirements specification.
4. Have analysts explain all work products created from the requirements process.
5. Expect analysts and developers to treat you with respect and to maintain a collaborative and professional attitude throughout your interactions.
6. Have analysts and developers provide ideas and alternatives both for your requirements and for implementation of the product.
7. Describe characteristics of the product that will make it easy and enjoyable to use.
8. Be given opportunities to adjust your requirements to permit reuse of existing software components.
9. Receive good-faith estimates of the costs, impacts, and trade-offs when you request a change in the requirements.

10. Receive a system that meets your functional and quality needs, to the extent that those needs have been communicated to the developers and agreed upon.

Table 2-2: Requirements Bill of Responsibilities for Software Customers

You have the responsibility to
1. Educate analysts and developers about your business and define business jargon.
2. Spend the time that it takes to provide requirements, clarify them, and iteratively flesh them out.
3. Be specific and precise when providing input about the system's requirements.
4. Make timely decisions about requirements when requested to do so.
5. Respect a developer's assessment of the cost and feasibility of requirements.
6. In collaboration with the developers, set priorities for functional requirements, system features, or use cases.
7. Review requirements documents and evaluate prototypes.
8. Communicate changes to the requirements as soon as you know about them.
9. Follow the development organization's process for requesting requirements changes.
10. Respect the processes the analysts use for requirements engineering.

These rights and responsibilities apply directly to customers when the software is being developed for internal corporate use, under contract, or for a known set of major customers. For mass-market product development, the rights and responsibilities are more applicable to customer surrogates such as the marketing department.

As part of project planning, the customer and development participants should review these two lists and reach a meeting of the minds. Busy customers might prefer not to become involved in requirements engineering (that is, they shy away from Responsibility #2). However, we know that lack of customer involvement greatly increases the risk of building the wrong product. Make sure the key participants in requirements development understand and accept their responsibilities. If you encounter some sticking points, negotiate to reach a clear understanding regarding your responsibilities to each other. This understanding can reduce friction later, when one party expects something that the other is not willing or able to provide.

Trap Don't assume that the project stakeholders instinctively know how to collaborate on requirements development. Take the time to discuss how you can work together most effectively.

Requirements Bill of Rights for Software Customers

Right #1: To Expect Analysts to Speak Your Language

Requirements discussions should center on your business needs and tasks, using your business vocabulary. Consider conveying business terminology to the analysts through a glossary. You shouldn't have to wade through computer jargon when talking with analysts.

Right #2: To Have Analysts Learn About Your Business and Objectives

By interacting with you to elicit requirements, the analysts can better understand your business tasks and how the product fits into your world. This will help developers create a system that satisfies your expectations. Consider inviting developers and analysts to observe what you and your colleagues do on the job. If the system being built is replacing an existing application, the developers should use the current system as you use it. This will help them see how the current application fits into your workflow and where it can be improved.

Right #3: To Expect Analysts to Write a Software Requirements Specification

The analyst will sort through all the information that you and other customers provide to distinguish use cases from business requirements, business rules, functional requirements, quality goals, solution ideas, and other items. The ultimate deliverable from this analysis is a software requirements specification (SRS), which is the agreement between developers and customers on the functions, qualities, and constraints of the product to be built. The SRS should be organized and written in a way that you find easy to understand. Your review of these specifications and other requirements representations helps to ensure that they accurately and completely represent your needs.

Right #4: To Receive Explanations of Requirements Work Products

The analyst might represent the requirements using various diagrams that complement the textual SRS. Chapter 11, "A Picture Is Worth 1024 Words," describes several such analysis models. These alternative views of the requirements are valuable because sometimes graphics are a clearer medium for expressing some aspects of system behavior, such as workflow. Although these diagrams might be unfamiliar, they aren't difficult to understand. Ask the analyst to explain the purpose of each diagram (and any other requirements development work products), what the notations mean, and how to examine the diagram for errors.

Right #5: To Expect Analysts and Developers to Treat You with Respect

Requirements discussions can be frustrating if customers and developers don't understand each other. Working together can open the eyes of both groups to the problems each group faces. Customers who participate in the requirements development process have the right to expect analysts and developers to treat them with respect and to appreciate the time they are investing in project success. Similarly, customers should demonstrate respect for the development team members as they all collaborate toward their mutual objective of a successful project.

Right #6: To Hear Ideas and Alternatives for Requirements and Their Implementation

Analysts should know about ways that your existing systems don't fit well with your business processes to make sure the new system doesn't automate ineffective or obsolete processes. Analysts who thoroughly understand the business domain can sometimes suggest improvements in your business processes. A creative analyst also adds value by proposing ways that new software can provide capabilities that customers haven't even envisioned.

Right #7: To Describe Characteristics That Make the Product Easy to Use

You can expect analysts to ask you about characteristics of the software that go beyond the user's functional needs. These characteristics—quality attributes— make the software easier or more pleasant to use, which lets users accomplish their tasks more efficiently. Users sometimes request that the product be *user-friendly* or *robust* or *efficient*, but such terms are too subjective to help the developers. Instead, the analysts should inquire about the specific characteristics that mean user-friendly, robust, or efficient to you. See Chapter 12, "Beyond Functionality: Software Quality Attributes," for further discussion.

Right #8: To Be Given Opportunities to Adjust Requirements to Permit Reuse

Requirements are often flexible. The analyst might know of existing software components that come close to addressing some need you described. In such a case, the analyst should give you the option of modifying your requirements so that the developers can reuse some existing software. Adjusting your requirements when sensible reuse opportunities are available saves time and money. Some requirements flexibility is essential if you want to incorporate commercial off-the-shelf (COTS) components into your product, because they will rarely have precisely the characteristics you want.

Right #9: To Receive Good-Faith Estimates of the Costs of Changes

People make different choices when they know that one alternative is more expensive than another. Estimates of the impact and cost of a proposed change in the requirements are necessary to make good business decisions about which requested changes to approve. You have the right to expect developers to present realistic estimates of impact, cost, and trade-offs. Developers must not inflate the estimated cost of a change just because they don't want to implement it.

Right #10: To Receive a System That Meets Your Functional and Quality Needs

Everyone desires this project outcome, but it can happen only if you clearly communicate all the information that will let developers build the right product and if developers clearly communicate options and constraints. Be sure to state all your assumptions or expectations; otherwise, the developers probably can't address them to your satisfaction.

Requirements Bill of Responsibilities for Software Customers

Responsibility #1: To Educate Analysts and Developers About Your Business

The development team depends on you to educate them about your business concepts and terminology. The intent is not to transform analysts into domain experts, but to help them understand your problems and objectives. Don't expect analysts to grasp the nuances and implicit aspects of your business. Analysts aren't likely to be aware of knowledge that you and your peers take for granted. Unstated assumptions about such knowledge can lead to problems later on.

Responsibility #2: To Spend the Time to Provide and Clarify Requirements

Customers are busy people and those who are involved in developing requirements are often among the busiest. Nonetheless, you have a responsibility to invest time in workshops, brainstorming sessions, interviews, and other requirements-elicitation activities. Sometimes the analyst might think she understands a point you made, only to realize later that she needs further clarification. Please be patient with this iterative approach to developing and refining the requirements; it's the nature of complex human communication and a key to software success. Be tolerant of what might appear to you to be dumb questions; a good analyst asks questions that get you talking.

Responsibility #3: To Be Specific and Precise About Requirements

It is tempting to leave the requirements vague and fuzzy because pinning down details is tedious and time consuming. At some point during development, though, someone must resolve the ambiguities and imprecisions. As the customer, you are the best person to make those decisions. Otherwise, you're relying on the developers to guess correctly.

It's fine to temporarily include *to be determined* (TBD) markers in the SRS to indicate that additional research, analysis, or information is needed. Sometimes, though, TBD is used because a specific requirement is difficult to resolve and no one wants to tackle it. Try to clarify the intent of each

requirement so that the analyst can express it accurately in the SRS. If you can't be precise, agree to a process to generate the necessary precision. This often involves some prototyping, in which you work with the developers in an incremental and iterative approach to requirements definition.

Responsibility #4: To Make Timely Decisions

Just as a contractor does when he's building your custom home, the analyst will ask you to make many choices and decisions. These decisions include resolving inconsistent requests received from multiple customers, choosing between conflicting quality attributes, and evaluating the accuracy of information. Customers who are authorized to make such decisions must do so promptly when asked. The developers often can't proceed with confidence until you render your decision, so time spent waiting for an answer can delay progress.

Responsibility #5: To Respect a Developer's Assessment of Cost and Feasibility

All software functions have a cost. Developers are in the best position to estimate those costs, although many developers are not skilled estimators. Some features that you want included might not be technically feasible or might be surprisingly expensive to implement. Certain requirements might demand unattainable performance in the operating environment or require access to data that is simply not available to the system. The developer can be the bearer of bad news about feasibility or cost, and you should respect that judgment.

Sometimes you can rewrite requirements in a way that makes them attainable or cheaper. For example, asking for an action to take place "instantaneously" isn't feasible, but a more specific timing requirement ("within 50 milliseconds") might be achievable.

Responsibility #6: To Set Requirement Priorities

Few projects have the time and resources to implement every bit of desirable functionality. Determining which capabilities are essential, which are useful, and which the customers can live without is an important part of requirements analysis. You have a lead role in setting those priorities because developers can't determine how important every requirement is to the customers. Developers will provide information about the cost and risk of each requirement to help determine final priorities. When you establish priorities, you help the developers deliver the maximum value at the lowest cost and at the right time.

Respect the development team's judgment as to how much of the requested functionality they can complete within the available time and resource constraints. No one likes to hear that something he or she wants can't be completed within the project bounds, but that's just a reality. The project's decision makers will have to elect whether to reduce project scope based on priorities, extend the schedule, provide additional funds or people, or compromise on quality.

Responsibility #7: To Review Requirements Documents and Evaluate Prototypes

As you'll see in Chapter 15, "Validating the Requirements," requirements reviews are among the most valuable software quality activities. Having customers participate in reviews is the only way to evaluate whether the requirements demonstrate the desired characteristics of being complete, correct, and necessary. A review also provides an opportunity for customer representatives to give the analysts feedback about how well their work is meeting the project's needs. If the representatives aren't confident that the documented requirements are accurate, they should tell the people responsible as early as possible and provide suggestions for improvement.

It's hard to develop a good mental picture of how the software will work by reading a requirements specification. To better understand your needs and explore the best ways to satisfy them, developers sometimes build prototypes of the intended product. Your feedback on these preliminary, partial, or

exploratory implementations provides valuable information to the developers. Recognize that a prototype is *not* a working product, and don't pressure the developers to deliver a prototype and pretend it is a fully functioning system.

Responsibility #8: To Promptly Communicate Changes to the Requirements

Continually changing requirements pose a serious risk to the development team's ability to deliver a high-quality product on schedule. Change is inevitable, but the later in the development cycle a change is introduced, the greater its impact. Changes can cause expensive rework, and schedules can slip if new functionality is demanded after construction is well under way. Notify the analyst with whom you are working as soon as you become aware that you need to change the requirements.

Responsibility #9: To Follow the Development Organization's Change Process

To minimize the negative impact of change, follow the project's defined change control process. This ensures that requested changes are not lost, the impact of each requested change is analyzed, and all proposed changes are considered in a consistent way. As a result, the business stakeholders can make sound business decisions to incorporate appropriate changes.

Responsibility #10: To Respect the Requirements Engineering Processes the Analysts Use

Gathering and validating requirements are among the greatest challenges in software development. There is a rationale behind the approaches that the analysts use. Although you might become frustrated with the requirements activities, the time spent understanding requirements is an excellent investment. The process will be less painful if you understand and respect the techniques the analysts use for requirements development. Feel free to ask analysts to explain why they're requesting certain information or asking you to participate in some requirements-related activity.

What About Sign-Off?

Reaching agreement on the requirements for the product to be built is at the core of the customer-developer partnership. Many organizations use the concept of signing off (why not "signing on"?) on the requirements document as the mark of customer approval of those requirements. All participants in the requirements approval process should know exactly what sign-off means or problems could ensue. One such problem is the customer representative who regards signing off on the requirements as a meaningless ritual: "I was presented with a piece of paper that had my name typed on it below a line, so I signed on the line because otherwise the developers wouldn't start coding." This attitude can lead to future conflicts when that customer wants to change the requirements or when he's surprised by what is delivered: "Sure, I signed off on the requirements, but I didn't have time to read them all. I trusted you guys—you let me down!"

Equally problematic is the development manager who views sign-off as a way to freeze the requirements. Whenever a change request is presented, he can point to the SRS and protest, "But you signed off on these requirements, so that's what we're building. If you wanted something else, you should have said so."

Both of these attitudes ignore the reality that it's impossible to know all the requirements early in the project and that requirements will undoubtedly change over time. Approving the requirements is an appropriate action that brings closure to the requirements development process. However, the participants have to agree on precisely what they're saying with their signatures.

Trap Don't use sign-off as a weapon. Use it as a project milestone, with a clear, shared understanding of the activities that lead to sign-off and its implications for future changes.

More important than the sign-off ritual is the concept of establishing a *baseline* of the requirements agreement, a snapshot of it at a point in time. The subtext of a signature on a requirements specification sign-off page should therefore read something like this: "I agree that this document represents our best understanding of the requirements for this project today and that the system described will satisfy our needs. I agree to make future changes in this baseline through the project's defined change process. I realize that approved changes might require us to renegotiate the cost, resource, and schedule commitments for this project." After the analyst defines the baseline, he places the requirements under change control. This allows the team to modify the project's scope when necessary in a controlled way that includes analyzing the impact of each proposed change on the schedule and on other project success factors.

A shared understanding along this line helps reduce the friction that can arise as requirements oversights are revealed or marketplace and business demands evolve in the course of the project. If customers are afraid that they won't be able to make changes after they approve the SRS, they might delay the approval, which contributes to the dreaded trap of analysis paralysis. A meaningful baselining process gives all the major stakeholders confidence in the following ways:

- Customer management is confident that the project scope won't explode out of control, because customers manage the scope change decisions.
- User representatives have confidence that development will work with them to deliver the right system, even if the representatives didn't think of every requirement before construction began.
- Development management has confidence because the development team has a business partner who will keep the project focused on achieving its objectives and will work with development to balance schedule, cost, functionality, and quality.
- Requirements analysts are confident because they know that they can manage changes to the project in a way that will keep chaos to a minimum.

Sealing the initial requirements development activities with such an explicit agreement helps you forge a collaborative customer-development partnership on the way to project success.

Next Steps

- Identify the customers who are responsible for providing the business and user requirements on your project. Which items from the Bill of Rights and the Bill of Responsibilities (on page 32) do these customers understand, accept, and practice? Which do they not?
- Discuss the Bill of Rights with your key customers to learn whether they feel they aren't receiving any of their rights. Discuss the Bill of Responsibilities to reach agreement as to which responsibilities they accept. Modify the Bill of Rights and the Bill of Responsibilities as appropriate so that all parties agree on how they will work together.
- If you're a customer participating in a software development project and you don't feel that your requirements rights are being adequately respected, discuss the Bill of Rights with the software project manager or the requirements analyst. Offer to do your part to satisfy the Bill of Responsibilities as you strive to build a more collaborative working relationship.
- Write a definition of what sign-off really means for your requirements documents approval.

Chapter 3: Good Practices for Requirements Engineering

Overview

Ten or fifteen years ago, I was a fan of software development methodologies—packaged sets of models and techniques that purport to provide holistic solutions to our project challenges. Today, though, I prefer to identify and apply industry best practices. Rather than devising or purchasing a whole-cloth solution, the best-practice approach stocks your software tool kit with a variety of techniques you can apply to diverse problems. Even if you do adopt a commercial methodology, adapt it to best suit your needs and augment its components with other effective practices from your tool kit.

The notion of best practices is debatable: who decides what is "best" and on what basis? One approach is to convene a body of industry experts or researchers to analyze projects from many different organizations (Brown 1996; Brown 1999; Dutta, Lee, and Van Wassenhove 1999). These experts look for practices whose effective performance is associated with successful projects and which are performed poorly or not at all on failed projects. Through these means, the experts reach consensus on the activities that consistently yield superior results. Such activities are dubbed *best practices*. This implies that they represent highly effective ways for software professionals to increase the chance of success on certain kinds of projects and in certain situations.

Table 3-1 lists nearly 50 practices, grouped into seven categories, that can help most development teams do a better job on their requirements activities. Several of the practices contribute to more than one category, but each practice appears only once in the table. These practices aren't suitable for every situation, so use good judgment, common sense, and experience instead of ritualistically following a script. Note that not all of these items have been endorsed as industry best practices, which is why I've titled this chapter "Good Practices for Requirements Engineering," not "Best Practices." I doubt whether all of these practices will ever be systematically evaluated for this purpose. Nonetheless, many other practitioners and I have found these techniques to be effective (Sommerville and Sawyer 1997; Hofmann and Lehner 2001). Each practice is described briefly in this chapter, and references are provided to other chapters in this book or to other sources where you can learn more about the technique. The last section of this chapter suggests a requirements development process—a sequence of activities—that is suitable for most software projects.

Table 3-1: Requirements Engineering Good Practices

Knowledge	Requirements Management	Project Management
• Train requirements analysts • Educate user reps and managers about requirements • Train developers in application domain • Create a glossary	• Define change-control process • Establish change control board • Perform change impact analysis • Baseline and control	• Select appropriate life cycle • Base plans on requirements • Renegotiate commitments • Manage requirements

	versions of requirements • Maintain change history • Track requirements status • Measure requirements volatility • Use a requirements management tool • Create requirements traceability matrix	risks • Track requirements effort • Review past lessons learned

Requirements Development

Elicitation	*Analysis*	*Specification*	*Validation*
• Define requirements development process • Define vision and scope • Identify user classes • Select product champions • Establish focus groups • Identify use cases • Identify system events and responses • Hold facilitated elicitation workshops • Observe users performing their jobs • Examine problem reports • Reuse requirements	• Draw context diagram • Create prototypes • Analyze feasibility • Prioritize requirements • Model the requirements • Create a data dictionary • Allocate requirements to subsystems • Apply Quality Function Deployment	• Adopt SRS template • Identify sources of requirements • Uniquely label each requirement • Record business rules • Specify quality attributes	• Inspect requirements documents • Test the requirements • Define acceptance criteria

PREVIOUS NEXT
PREVIOUS NEXT

Knowledge

Few software developers receive formal training in requirements engineering. However, many

developers perform the role of requirements analyst at some point in their careers, working with customers to gather, analyze, and document requirements. It isn't reasonable to expect all developers to be instinctively competent at the communication-intensive tasks of requirements engineering. Training can increase the proficiency and comfort level of those who serve as analysts, but it can't compensate for missing interpersonal skills or a lack of interest.

Because the requirements process is essential, all project stakeholders should understand the concepts and practices of requirements engineering. Bringing together the various stakeholders for a one-day overview on software requirements can be an effective team-building activity. All parties will better appreciate the challenges their counterparts face and what the participants require from each other for the whole team to succeed. Similarly, developers should receive grounding in the concepts and terminology of the application domain. You can find further details on these topics in the following chapters:

- Chapter 4—Train requirements analysts.
- Chapter 10—Create a project glossary.

Train requirements analysts. All team members who will function as analysts should receive basic training in requirements engineering. Requirements analyst specialists need several days of training in these activities. The skilled requirements analyst is patient and well organized, has effective interpersonal and communication skills, understands the application domain, and has an extensive tool kit of requirements-engineering techniques.

True Stories **Educate user representatives and managers about software requirements.** Users who will participate in software development should receive one or two days of education about requirements engineering. Development managers and customer managers will also find this information useful. The training will help them understand the value of emphasizing requirements, the activities and deliverables involved, and the risks of neglecting requirements processes. Some users who have attended my requirements seminars have said that they came away with more sympathy for the software developers.

Train developers in application domain concepts. To help developers achieve a basic understanding of the application domain, arrange a seminar on the customer's business activities, terminology, and objectives for the product being created. This can reduce confusion, miscommunication, and rework down the road. You might also match each developer with a "user buddy" for the life of the project to translate jargon and explain business concepts. The product champion could play this role.

Create a project glossary. A glossary that defines specialized terms from the application domain will reduce misunderstandings. Include synonyms, terms that can have multiple meanings, and terms that have both domain-specific and everyday meanings. Words that can be both nouns and verbs—such as "process" and "order"—can be particularly confusing.

PREVIOUS NEXT
PREVIOUS NEXT

Requirements Elicitation

Chapter 1 discussed the three levels of requirements: business, user, and functional. These come from different sources at different times during the project, have different audiences and purposes, and need to be documented in different ways. The business requirements expressed in the project scope must not exclude any essential user requirements, and you should be able to trace all functional

requirements back to specific user requirements. You also need to elicit nonfunctional requirements, such as quality and performance expectations, from appropriate sources. You can find additional information about these topics in the following chapters:

- Chapter 3—Define a requirements development process.
- Chapter 5—Write a vision and scope document.
- Chapter 6—Identify user classes and their characteristics; select a product champion for each user class; observe users performing their jobs.
- Chapter 7—Hold facilitated elicitation workshops.
- Chapter 8—Work with user representatives to identify use cases; identify system events and responses.
- Chapter 22—Define a requirements development process.

Define a requirements development process. Document the steps your organization follows to elicit, analyze, specify, and validate requirements. Providing guidance on how to perform the key steps will help analysts do a consistently good job. It will also make it easier to plan each project's requirements development tasks, schedule, and required resources.

Write a vision and scope document. The vision and scope document contains the product's business requirements. The vision statement gives all stakeholders a common understanding of the product's objectives. The scope defines the boundary between what's in and what's out for a specific release. Together, the vision and scope provide a reference against which to evaluate proposed requirements. The product vision should remain relatively stable from release to release, but each release needs its own project scope statement.

Identify user classes and their characteristics. To avoid overlooking the needs of any user community, identify the various groups of users for your product. They might differ in frequency of use, features used, privilege levels, or skill levels. Describe aspects of their job tasks, attitudes, location, or personal characteristics that might influence product design.

Select a product champion for each user class. Identify at least one person who can accurately serve as the voice of the customer for each user class. The product champion presents the needs of the user class and makes decisions on its behalf. This is easiest for internal information systems development, where your users are fellow employees. For commercial development, build on your current relationships with major customers or beta test sites to locate appropriate product champions. Product champions must have ongoing participation in the project and the authority to make decisions at the user-requirements level.

Establish focus groups of typical users. Convene groups of representative users of your previous product releases or of similar products. Collect their input on both functionality and quality characteristics for the product under development. Focus groups are particularly valuable for commercial product development, for which you might have a large and diverse customer base. Unlike product champions, focus groups generally do not have decision-making authority.

Work with user representatives to identify use cases. Explore with your user representatives the tasks they need to accomplish with the software—their use cases. Discuss the interactions between the users and the system that will allow them to complete each such task. Adopt a standard template for documenting use cases and derive functional requirements from those use cases. A related practice that is often used on government projects is to define a concept of operations (ConOps) document, which describes the new system's characteristics from the user's point of view (IEEE 1998a).

Identify system events and responses. List the external events that the system can experience and its expected response to each event. Events include signals or data received from external hardware devices and temporal events that trigger a response, such as an external data feed that your system generates at the same time every night. Business events trigger use cases in business applications.

Hold facilitated elicitation workshops. Facilitated requirements-elicitation workshops that permit collaboration between analysts and customers are a powerful way to explore user needs and to draft requirements documents (Gottesdiener 2002). Specific examples of such workshops include Joint Requirements Planning (JRP) sessions (Martin 1991) and Joint Application Development (JAD) sessions (Wood and Silver 1995).

Observe users performing their jobs. Watching users perform their business tasks establishes a context for their potential use of a new application (Beyer and Holtzblatt 1998). Simple workflow diagrams—data flow diagrams work well—can depict when the user has what data and how that data is used. Documenting the business process flow will help you identify requirements for a system that's intended to support that process. You might even determine that the customers don't really need a new software application to meet their business objectives (McGraw and Harbison 1997).

Examine problem reports of current systems for requirement ideas. Problem reports and enhancement requests from customers provide a rich source of ideas for capabilities to include in a later release or in a new product. Help desk and support staff can provide valuable input to the requirements for future development work.

Reuse requirements across projects. If customers request functionality similar to that already present in an existing product, see whether the requirements (and the customers!) are flexible enough to permit reusing or adapting the existing components. Multiple projects will reuse those requirements that comply with an organization's business rules. These include security requirements that control access to the applications and requirements that conform to government regulations, such as the Americans with Disabilities Act.

PREVIOUS NEXT
PREVIOUS NEXT

Requirements Analysis

Requirements analysis involves refining the requirements to ensure that all stakeholders understand them and scrutinizing them for errors, omissions, and other deficiencies. Analysis includes decomposing high-level requirements into details, building prototypes, evaluating feasibility, and negotiating priorities. The goal is to develop requirements of sufficient quality and detail that managers can construct realistic project estimates and technical staff can proceed with design, construction, and testing.

Often it is helpful to represent some of the requirements in multiple ways—for example, in both textual and graphical forms. These different views will reveal insights and problems that no single view can provide (Davis 1995). Multiple views also help all stakeholders arrive at a common understanding—a shared vision—of what they will have when the product is delivered. Further discussion of requirements analysis practices can be found in the following chapters:

- Chapter 5—Draw a context diagram.
- Chapter 10—Create a data dictionary.
- Chapter 11—Model the requirements.

- Chapter 13—Create user interface and technical prototypes.
- Chapter 14—Prioritize the requirements.
- Chapter 17—Allocate requirements to subsystems.

Draw a context diagram. The context diagram is a simple analysis model that shows how the new system fits into its environment. It defines the boundaries and interfaces between the system being developed and the entities external to the system, such as users, hardware devices, and other information systems.

Create user interface and technical prototypes. When developers or users aren't certain about the requirements, construct a prototype—a partial, possible, or preliminary implementation—to make the concepts and possibilities more tangible. Users who evaluate the prototype help the stakeholders achieve a better mutual understanding of the problem being solved.

Analyze requirement feasibility. Evaluate the feasibility of implementing each requirement at acceptable cost and performance in the intended operating environment. Understand the risks associated with implementing each requirement, including conflicts with other requirements, dependencies on external factors, and technical obstacles.

Prioritize the requirements. Apply an analytical approach to determine the relative implementation priority of product features, use cases, or individual requirements. Based on priority, determine which release will contain each feature or set of requirements. As you accept requirement changes, allocate each one to a particular future release and incorporate the effort required to make the change into the plan for that release. Evaluate and adjust priorities periodically throughout the project as customer needs, market conditions, and business goals evolve.

Model the requirements. A graphical analysis model depicts requirements at a high level of abstraction, in contrast to the detail shown in the SRS or the user interface view that a prototype provides. Models can reveal incorrect, inconsistent, missing, and superfluous requirements. Such models include data flow diagrams, entity-relationship diagrams, state-transition diagrams or statecharts, dialog maps, class diagrams, sequence diagrams, interaction diagrams, decision tables, and decision trees.

Create a data dictionary. Definitions of all the data items and structures associated with the system reside in the data dictionary. This enables everyone working on the project to use consistent data definitions. At the requirements stage, the data dictionary should define data items from the problem domain to facilitate communication between the customers and the development team.

Allocate requirements to subsystems. The requirements for a complex product that contains multiple subsystems must be apportioned among the various software, hardware, and human subsystems and components (Nelsen 1990). A system engineer or architect typically performs this allocation.

Apply Quality Function Deployment. Quality Function Deployment (QFD) is a rigorous technique for relating product features and attributes to customer value (Zultner 1993; Pardee 1996). It provides an analytical way to identify those features that will provide the greatest customer satisfaction. QFD addresses three classes of requirements: expected requirements, where the customer might not even state them but will be upset if they are missing; normal requirements; and exciting requirements, which provide high benefit to customers if present but little penalty if not.

Requirements Specification

No matter how you obtain your requirements, document them in some consistent, accessible, and reviewable way. You can record the business requirements in a vision and scope document. The user requirements typically are represented in the form of use cases or as event-response tables. The SRS contains the detailed software functional and nonfunctional requirements. Requirements specification practices are discussed in the following chapters:

- Chapter 9—Record business rules.
- Chapter 10—Adopt an SRS template; uniquely label each requirement.
- Chapter 12—Specify quality attributes.

Adopt an SRS template. Define a standard template for documenting software requirements in your organization. The template provides a consistent structure for recording the functionality descriptions and other requirements-related information. Rather than inventing a new template, adapt an existing one to fit the nature of your projects. Many organizations begin with the SRS template described in IEEE Standard 830-1998 (IEEE 1998b); Chapter 10 presents an adaptation of that template. If your organization works on different types or sizes of projects, such as large new development efforts as well as small enhancement releases, define an appropriate template for each project type. Templates and processes should both be scalable.

Identify sources of requirements. To ensure that all stakeholders know why every requirement belongs in the SRS and to facilitate further clarification, trace each requirement back to its origin. This might be a use case or some other customer input, a high-level system requirement, a business rule, or some other external source. Recording the stakeholders who are materially affected by each requirement tells you whom to contact when a change request comes in. Requirement sources can be identified either through traceability links or by defining a requirement attribute for this purpose. See Chapter 18 for more information on requirements attributes.

Uniquely label each requirement. Define a convention for giving each requirement in the SRS a unique identifying label. The convention must be robust enough to withstand additions, deletions, and changes made in the requirements over time. Labeling the requirements permits requirements traceability and the recording of changes made.

Record business rules. Business rules include corporate policies, government regulations, and computational algorithms. Document your business rules separately from the SRS because they typically have an existence beyond the scope of a specific project. Some business rules will lead to functional requirements that enforce them, so define traceability links between those requirements and the corresponding rules.

Specify quality attributes. Go beyond the functionality discussion to explore quality characteristics that will help your product satisfy customer expectations. These characteristics include performance, efficiency, reliability, usability, and many others. Document the quality requirements in the SRS. Customer input on the relative importance of these quality attributes lets the developer make appropriate design decisions.

PREVIOUS NEXT
PREVIOUS NEXT

Requirements Validation

Validation ensures that the requirement statements are correct, demonstrate the desired quality characteristics, and will satisfy customer needs. Requirements that seem fine when you read them in the SRS might turn out to have problems when developers try to work with them. Writing test cases from the requirements often reveals ambiguities and vagueness. You must correct these problems if the requirements are to serve as a reliable foundation for design and for final system verification through system testing or user acceptance testing. Requirements validation is discussed further in Chapter 15.

Inspect requirements documents. Formal inspection of requirements documents is one of the highest-value software quality practices available. Assemble a small team of inspectors who represent different perspectives (such as analyst, customer, developer, and tester), and carefully examine the SRS, analysis models, and related information for defects. Informal preliminary reviews during requirements development are also valuable. Even though this is not one of the easiest new practices to implement, it's among the most valuable, so begin building requirements inspections into your culture right away.

Test the requirements. Derive functional test cases from the user requirements to document the expected behavior of the product under specified conditions. Walk through the test cases with customers to ensure that they reflect the desired system behavior. Trace the test cases to the functional requirements to make sure that no requirements have been overlooked and that all have corresponding test cases. Use the test cases to verify the correctness of analysis models and prototypes.

Define acceptance criteria. Ask users to describe how they will determine whether the product meets their needs and is fit for use. Base acceptance tests on usage scenarios (Hsia, Kung, and Sell 1997).

Requirements Management

Once you have the initial requirements for a body of work in hand, you must cope with the inevitable changes that customers, managers, marketing, the development team, and others request during development. Effective change management demands a process for proposing changes and evaluating their potential cost and impact on the project. A *change control board* (CCB), composed of key stakeholders, decides which proposed changes to incorporate. Tracking the status of each requirement as it moves through development and system testing provides insight into overall project status.

Well-established configuration management practices are a prerequisite for effective requirements management. The same version control tools that you use to control your code base can manage your requirements documents. The techniques involved in requirements management are expanded in the following chapters:

- Chapter 18—Establish a baseline and control versions of requirements; track the status of each requirement
- Chapter 19—Define a requirements change-control process; establish a change control board; measure requirements volatility; perform requirements-change impact analysis
- Chapter 20—Create a requirements traceability matrix
- Chapter 21—Use a requirements management tool

Define a requirements change-control process. Establish a process through which requirements changes are proposed, analyzed, and resolved. Manage all proposed changes through this process. Commercial defect-tracking tools can support the change-control process.

Establish a change control board. Charter a small group of project stakeholders as a CCB to receive proposed requirements changes, evaluate them, decide which ones to accept and which to reject, and set implementation priorities or target releases.

Perform requirements-change impact analysis. Impact analysis helps the CCB make informed business decisions. Evaluate each proposed requirement change to determine the effect it will have on the project. Use the requirements traceability matrix to identify the other requirements, design elements, source code, and test cases that you might have to modify. Identify the tasks required to implement the change and estimate the effort needed to perform those tasks.

Establish a baseline and control versions of requirements documents. The baseline consists of the requirements that have been committed to implementation in a specific release. After the requirements have been baselined, changes may be made only through the defined change-control process. Give every version of the requirements specification a unique identifier to avoid confusion between drafts and baselines and between previous and current versions. A more robust solution is to place requirements documents under version control using appropriate configuration management tools.

Maintain a history of requirements changes. Record the dates that requirements specifications were changed, the changes that were made, who made each change, and why. A version control tool or commercial requirements management tool can automate these tasks.

Track the status of each requirement. Establish a database with one record for each discrete functional requirement. Store key attributes about each requirement, including its status (such as proposed, approved, implemented, or verified), so that you can monitor the number of requirements in each status category at any time.

Measure requirements volatility. Record the number of baselined requirements and the number of proposed and approved changes (additions, modifications, deletions) to them per week. Churning requirements might suggest that the problem is not well understood, the project scope is not well defined, the business is changing rapidly, many requirements were missed during elicitation, or politics are running rampant.

Use a requirements management tool. Commercial requirements management tools let you store various types of requirements in a database. You can define attributes for each requirement, track each requirement's status, and define traceability links between requirements and other software work products. This practice will help you automate the other requirements management tasks described in this section.

Create a requirements traceability matrix. Set up a table that links each functional requirement to the design and code elements that implement it and the tests that verify it. The requirements traceability matrix can also connect functional requirements to the higher-level requirements from which they were derived and to other related requirements. Populate this matrix during development, not at the end of the project.

PREVIOUS NEXT
PREVIOUS NEXT

Project Management

Software project management approaches are intimately related to a project's requirements processes. Base your project resources, schedules, and commitments on the requirements that are to be implemented. Because changes in requirements will affect those project plans, the plans should anticipate some requirements change and scope growth (Wiegers 2002d). More information about project management approaches to requirements engineering is available in the following chapters:

- Chapter 17—Base project plans on requirements
- Chapter 18—Track the effort spent on requirements engineering
- Chapter 23—Document and manage requirements-related risks

Select an appropriate software development life cycle. Your organization should define several development life cycles that are appropriate for various types of projects and different degrees of requirements uncertainty (McConnell 1996). Each project manager should select and adapt the life cycle that best suits his project. Include requirements engineering activities in your life cycle definitions. If the requirements or scope are poorly defined early in the project, plan to develop the product in small increments, beginning with the most clearly understood requirements and a robust, modifiable architecture. When possible, implement sets of features so that you can release portions of the product periodically and deliver value to the customer as early as possible (Gilb 1988; Cockburn 2002).

Base project plans on requirements. Develop plans and schedules for your project iteratively as the scope and detailed requirements become clear. Begin by estimating the effort needed to develop the functional requirements from the initial product vision and project scope. Early cost and schedule estimates based on fuzzy requirements will be highly uncertain, but you can improve the estimates as your understanding of the requirements improves.

Renegotiate project commitments when requirements change. As you incorporate new requirements into the project, evaluate whether you can still achieve the current schedule and quality commitments with the available resources. If not, communicate the project realities to management and negotiate new, realistically achievable commitments (Humphrey 1997; Fisher, Ury, and Patton 1991; Wiegers 2002b). If your negotiations are unsuccessful, communicate the likely outcomes so that managers and customers aren't blindsided by an unexpected project outcome.

Document and manage requirements-related risks. Identify and document risks related to requirements as part of the project's risk management activities. Brainstorm approaches to mitigate or prevent these risks, implement the mitigation actions, and track their progress and effectiveness.

Track the effort spent on requirements engineering. Record the effort your team expends on requirements development and management activities. Use this data to assess whether the planned requirements activities are being performed as intended and to better plan the resources needed for future projects. In addition, monitor the effect that your requirements engineering activities have on the project. This will help you judge the return on your investment in requirements engineering.

Review lessons learned regarding requirements on other projects. A learning organization conducts project *retrospectives*—also called *postmortems* and *post-project reviews*—to collect lessons learned (Robertson and Robertson 1999; Kerth 2001; Wiegers and Rothman 2001). Studying the lessons learned about requirements issues and practices on previous projects can help project managers and requirements analysts steer a more confident course in the future.

PREVIOUS NEXT
PREVIOUS NEXT

Getting Started with New Practices

Table 3-2 groups the requirements engineering practices described in this chapter by the relative impact they can have on most projects and their relative difficulty of implementation. Although all the practices can be beneficial, you might begin with the "low-hanging fruit," those practices that have a high impact on project success and are relatively easy to implement.

Table 3-2: Implementing Requirements Engineering Good Practices

Impact			**Difficulty**
	High	***Medium***	***Low***
High	• Define requirements development process • Base plans on requirements • Renegotiate commitments	• Identify use cases • Specify quality attributes • Prioritize requirements • Adopt SRS template • Define change-control process • Establish CCB • Inspect requirements documents • Allocate requirements to subsystems • Record business rules	• Train developers in application domain • Define vision and scope • Identify user classes • Draw context diagram • Identify sources of requirements • Baseline and control versions of requirements
Medium	• Educate user reps and managers about requirements • Model the requirements • Manage requirements risks • Use a requirements management tool • Create requirements traceability matrix • Hold facilitated elicitation workshops	• Train requirements analysts • Select product champions • Establish focus groups • Create prototypes • Define acceptance criteria • Perform change impact analysis • Select appropriate life cycle	• Analyze feasibility • Create a glossary • Create a data dictionary • Observe users performing their jobs • Identify system events and responses • Uniquely label each requirement • Test the requirements

			• Track requirements status • Review past lessons learned
Low	• Reuse requirements • Apply Quality Function Deployment • Maintain change history volatility	• Measure requirements • Track requirements effort	• Examine problem reports

Don't try to apply all these techniques on your next project. Instead, think of these good practices as new items for your requirements tool kit. You can begin to use some practices, such as those dealing with change management, no matter where your project is in its development cycle. Elicitation practices will be more useful when you begin the next project or iteration. Still others might not fit your current project, organizational culture, or resource availability. Chapter 22 describes ways to evaluate your current requirements engineering practices. It will also help you devise a road map for implementing selected improvements in your requirements process based on the practices described in this chapter.

PREVIOUS NEXT
PREVIOUS NEXT

A Requirements Development Process

Don't expect to perform the requirements development activities of elicitation, analysis, specification, and validation in a linear, one-pass sequence. In practice, these activities are interleaved, incremental, and iterative, as shown in Figure 3-1. As you work with customers in your analyst role, you'll be asking questions, listening to what the customers say, and watching what they do (elicitation). You'll process this information to understand it, classify it in various categories, and relate the customer needs to possible software requirements (analysis). You'll then structure the customer input and derived requirements as written documents and diagrams (specification). Next, you'll ask your customer representatives to confirm that what you've written is accurate and complete and to correct any errors (validation). This iterative process continues throughout requirements development.

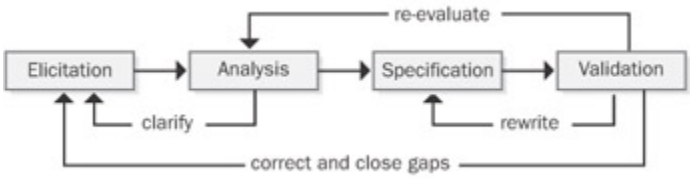

Figure 3-1: Requirements development is an iterative process.

Because of the diversity of software development projects and organizational cultures, there is no single, formulaic approach to requirements development. Figure 3-2 suggests a process framework for requirements development that will work—with sensible adjustments—for many projects. These steps are generally performed approximately in numerical sequence, but the process is not strictly sequential. The first seven steps are typically performed once early in the project (although the team will need to revisit priorities periodically). The remaining steps are performed for each release

increment or iteration.

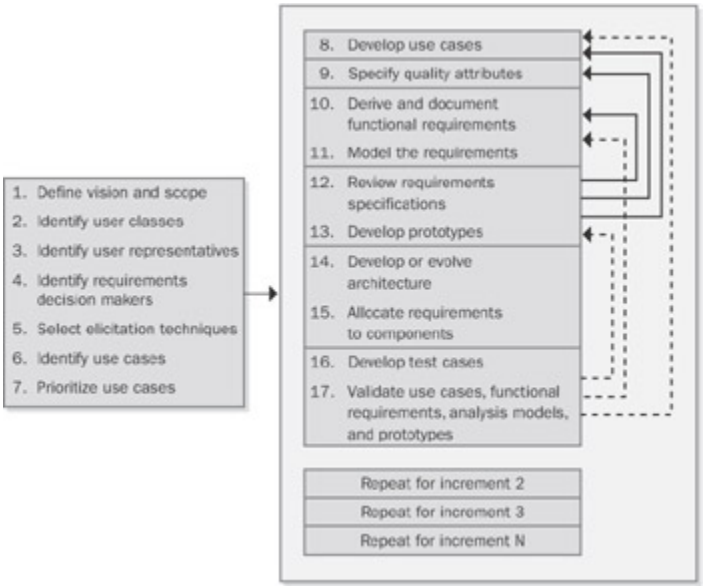

Figure 3-2: A suggested requirements development process, showing quality control feedback cycles and incremental implementation based on use-case priorities.

Select appropriate elicitation techniques (workshops, surveys, interviews, and so on) based on your access to the user representatives, and plan the time and resources that elicitation will consume (step 5 in Figure 3-2). Because many systems are built incrementally, every project team needs to prioritize its use cases or other user requirements (step 7). Prioritizing the use cases lets you decide which ones to plan for each increment so that you can explore the right use cases in detail at the right time. For new systems or major enhancements, you can define or refine the architecture in step 14 and allocate functional requirements to specific subsystems in step 15. Steps 12 and 17 are quality control activities that might lead you to revisit some earlier steps to correct errors, refine analysis models, or detect previously overlooked requirements. Prototypes built in step 13 often lead to refinement and modifications in the requirements that were specified previously. Once you have completed step 17 for any portion of the requirements, you're ready to commence construction of that part of the system. Repeat steps 8 through 17 with the next set of use cases, which might go into a late release.

Next Steps

- Go back to the requirements-related problems you identified from the Next Steps in Chapter 1. Identify good practices from this chapter that might help with each problem you identified. The troubleshooting guide in Appendix C might be helpful. Group the practices into high, medium, and low impact in your organization. Identify any barriers to implementing each practice in your organization or culture. Who can help you break down those barriers?

- Determine how you would assess the benefits from the practices that you think would be most valuable. Would you find fewer requirements defects late in the game, reduce unnecessary rework, better meet project schedules, or enjoy other advantages?

- List all the requirements good practices you identified in the first step. For each practice, indicate your project team's current level of capability: expert, proficient, novice, or unfamiliar. If your team is not at least proficient in any of those practices, ask someone on your project to learn more about the practice and to share what he learns with the rest of the team.

PREVIOUS NEXT

Chapter 4: The Requirements Analyst

Overview

Explicitly or implicitly, someone performs the role of requirements analyst on every software project. Corporate IS organizations identify specialists called *business analysts* to perform this function. Synonyms for *requirements analyst* include *systems analyst*, *requirements engineer*, *requirements manager*, and simply *analyst*. In a product-development organization, the job is often the product manager's or marketing staff's responsibility. The analyst is a translator of others' perspectives into a requirements specification and a reflector of information back to other stakeholders. The analyst helps stakeholders find the difference between what they say they want and what they really need. He or she educates, questions, listens, organizes, and learns. It's a tough job.

This chapter looks at the vital functions the requirements analyst performs, the skills and knowledge an effective analyst needs, and how you might develop such people in your organization (Wiegers 2000). You can download a sample job description for a requirements analyst from *http://www.processimpact.com/goodies.shtml*.

PREVIOUS NEXT
PREVIOUS NEXT

The Requirements Analyst Role

The requirements analyst is the individual who has the primary responsibility to gather, analyze, document, and validate the needs of the project stakeholders. The analyst serves as the principal conduit through which requirements flow between the customer community and the software development team, as shown in Figure 4-1. Many other communication pathways also are used, so the analyst isn't solely responsible for information exchange on the project. The analyst plays a central role in collecting and disseminating product information, whereas the project manager takes the lead in communicating project information.

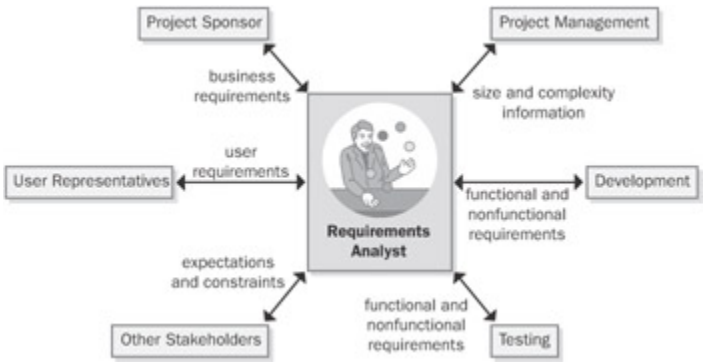

Figure 4-1: The requirements analyst bridges communication between customer and development stakeholders.

Requirements analyst is a project role, not necessarily a job title. One or more dedicated specialists could perform the role, or it could be assigned to team members who also have other job functions. These functions include project manager, product manager, subject matter expert (SME), developer, and even user. Regardless of their other project responsibilities, analysts must have the skills, knowledge, and personality to perform the analyst role well.

Trap Don't assume that any talented developer or knowledgeable user will automatically be an effective requirements analyst without training, resource materials, and coaching. These roles all demand different skills, knowledge, and personality traits.

True Stories A talented analyst can make the difference between a project that succeeds and one that struggles. One of my consulting clients discovered that they could inspect requirements specifications written by experienced analysts twice as fast as those written by novices because they contained fewer defects. In the widely-used Cocomo II model for project estimation, requirements analyst experience and capability have a great influence on a project's effort and cost (Boehm et al. 2000). Using highly experienced analysts can reduce the project's required effort by one third compared to similar projects with inexperienced analysts. Analyst capability has an even greater impact than analyst experience. Projects having the most capable analysts require only half the effort of similar projects that have the least capable analysts.

The Analyst's Tasks

The analyst is a communication middleman, bridging the gap between vague customer notions and the clear specifications that guide the software team's work. The analyst must first understand the users' goals for the new system and then define functional and quality requirements that allow project managers to estimate, developers to design and build, and testers to verify the product. This section describes some of the typical activities that you might perform while you're wearing an analyst's hat.

Define business requirements. Your work as an analyst begins when you help the business or funding sponsor, product manager, or marketing manager define the project's business requirements. Perhaps the first question to ask is, "Why are we undertaking this project?" Business requirements include a statement of the organization's business objectives and the ultimate vision of what the system will be and do. You might suggest a template for a vision and scope document (shown in Chapter 5) and work with those who hold the vision to help them express it.

Identify project stakeholders and user classes. The vision and scope document will help you identify the important user classes and other stakeholders for the product. Next, work with the business sponsors to select appropriate representatives for each user class, enlist their participation, and negotiate their responsibilities. User representatives might hesitate to participate in requirements exploration until they know exactly what you expect from them. Write down the contributions that you would like from your customer collaborators and agree on an appropriate level of participation from each one. Chapter 6 lists some activities that you might ask your user representatives to perform.

Elicit requirements. Requirements for a software product don't just lie around waiting for someone wearing a hat labeled "analyst" to collect them. A proactive analyst helps users articulate the system capabilities they need to meet their business objectives. See Chapter 7 and Chapter 8 for further discussion. You might employ information-gathering techniques selected from the following list:

- Interviews
- Facilitated requirements workshops
- Document analysis
- Surveys
- Customer site visits
- Business process analysis

- Work flow and task analysis
- Event lists
- Competitive product analysis
- Reverse engineering of existing systems
- Retrospectives performed on the previous project

Users naturally emphasize the system's functional requirements, so steer the discussions to include quality attributes, performance goals, business rules, external interfaces, and constraints. It's appropriate to challenge user assumptions, but don't try to force your own beliefs on the users. Some user requirements might seem absurd, but if the user confirms that they're correct, there's nothing to gain from pushing the point.

Analyze requirements. Look for derived requirements that are a logical consequence of what the customers requested and for unstated requirements that the customers seem to expect without saying so. Spot the vague, weak words that cause ambiguity. (See Chapter 10 for examples.) Point out conflicting requirements and areas that need more detail. Specify the functional requirements at a level of detail suitable for use by the developers who will implement them. This level of detail will vary from project to project. A Web site being built incrementally by a small, well-synchronized team can get away with limited requirements documentation. In contrast, a complex embedded system that will be outsourced to an offshore supplier needs a precise, detailed SRS.

Write requirements specifications. Requirements development leads to a shared understanding of a system that will address the customer's problem. The analyst is responsible for writing well-organized specifications that clearly express this shared understanding. Using standard templates for use cases and the SRS accelerates requirements development by reminding the analyst of topics that he needs to discuss with the user representatives. Chapter 8 discusses how to write use cases, Chapter 10 explores writing functional requirements, and Chapter 12 looks at documenting software quality attributes.

Model the requirements. The analyst should determine when it is helpful to represent requirements using methods other than text. These alternative views include various types of graphical analysis models (discussed in Chapter 11), tables, mathematical equations, storyboards, and prototypes (discussed in Chapter 13). Analysis models depict information at a higher level of abstraction than does detailed text. To maximize communication and clarity, draw analysis models according to the conventions of a standard modeling language.

Lead requirements validation. The analyst must ensure that requirement statements possess all the desired characteristics that were discussed in Chapter 1 and that a system based on the requirements will satisfy user needs. Analysts are the central participants in peer reviews of requirements documents. They should also review designs, code, and test cases that were derived from the requirements specifications to ensure that the requirements were interpreted correctly.

Facilitate requirements prioritization. The analyst brokers collaboration and negotiation between the various user classes and the developers to ensure that they make sensible priority decisions. The requirements prioritization spreadsheet tool described in Chapter 14 might be helpful.

Manage requirements. A requirements analyst is involved throughout the entire software development life cycle, so he should help create, review, and execute the project's requirements management plan. After establishing the requirements baseline, the analyst's focus shifts to managing those requirements and verifying their satisfaction in the product. Storing the requirements in a commercial requirements management tool facilitates this ongoing management. See Chapter 21 for a discussion of requirements management tools.

Requirements management includes tracking the status of individual functional requirements as they progress from inception to verification in the integrated product. With input from various colleagues, the analyst collects traceability information that connects individual requirements to other system elements. The analyst plays a central role in managing changes to the baselined requirements by using a change-control process and tool.

Essential Analyst Skills

It isn't reasonable to expect people to serve as analysts without sufficient training, guidance, and experience. They won't do a good job and they'll find the experience frustrating. Analysts need to know how to use a variety of elicitation techniques and how to represent information in forms other than natural-language text. An effective analyst combines strong communication, facilitation, and interpersonal skills with technical and business domain knowledge and the right personality for the job (Ferdinandi 2002). Patience and a genuine desire to work with people are key success factors. The skills described in the remainder of this section are particularly important.

Listening skills. To become proficient at two-way communication, learn how to listen effectively. Active listening involves eliminating distractions, maintaining an attentive posture and eye contact, and restating key points to confirm your understanding. You need to grasp what people are saying and also to read between the lines to detect what they might be hesitant to say. Learn how your collaborators prefer to communicate and avoid imposing your personal filter of understanding on what you hear the customers say. Watch for assumptions that underlie either what you hear from others or your own interpretation.

Interviewing and questioning skills. Most requirements input comes through discussions, so the analyst must be able to talk with diverse individuals and groups about their needs. It can be intimidating to work with senior managers and with highly opinionated or aggressive individuals. You need to ask the right questions to surface essential requirements information. For example, users naturally focus on the system's normal, expected behaviors. However, much code gets written to handle exceptions, so you must also probe for possible error conditions and determine how the system should respond. With experience, you'll become skilled in the art of asking questions that reveal and clarify uncertainties, disagreements, assumptions, and unstated expectations (Gause and Weinberg 1989).

Analytical skills. An effective analyst can think at multiple levels of abstraction. Sometimes you must drill down from high-level information into details. In other situations, you'll need to generalize from a specific need that one user described to a set of requirements that will satisfy many members of a user class. Critically evaluate the information gathered from multiple sources to reconcile conflicts, separate user "wants" from the underlying true needs, and distinguish solution ideas from requirements.

Facilitation skills. The ability to facilitate requirements elicitation workshops is a necessary analyst capability (Gottesdiener 2002). A neutral facilitator who has strong questioning, observational, and facilitation skills can help a group build trust and improve the sometimes tense relationship between business and information technology staff. Chapter 7 presents some guidelines for facilitating elicitation workshops.

Observational skills. An observant analyst will detect comments made in passing that might turn out to be significant. By watching a user perform his job or use a current application, a good observer can detect subtleties that the user might not mention. Strong observational skills sometimes expose new areas for elicitation discussions, revealing additional requirements that no one has mentioned yet.

Writing skills. The principal deliverable from requirements development is a written specification that communicates information among customers, marketing, managers, and technical staff. The analyst needs a solid command of the language and the ability to express complex ideas clearly.

True Stories I know of an organization that appointed as analysts several developers who spoke English as a second language. It's hard enough to write excellent requirements in your native language. It's even more difficult to accomplish in a language in which you might struggle with nuances of expression, ambiguous words, and local idioms. Conversely, analysts should be efficient and critical readers because they have to wade through a lot of material and grasp the essence quickly.

Organizational skills. Analysts must work with a vast array of jumbled information gathered during elicitation and analysis. Coping with rapidly changing information and structuring all the bits into a coherent whole demands exceptional organizational skills and the patience and tenacity to make sense from ambiguity and disarray.

Modeling skills. Tools ranging from the venerable flowchart through structured analysis models (data flow diagram, entity-relationship diagram, and the like) to contemporary Unified Modeling Language (UML) notations should be part of every analyst's repertoire. Some will be useful when communicating with users, others when communicating with developers. The analyst will need to educate other stakeholders on the value of using these techniques and how to read them. See Chapter 11 for overviews of several types of analysis models.

Interpersonal skills. Analysts need the ability to get people with competing interests to work together. An analyst should feel comfortable talking with individuals in diverse job functions and at all levels of the organization. He or she might need to work with distributed virtual teams whose members are separated by geography, time zones, cultures, or native languages. Experienced analysts often mentor their new colleagues, and they educate their customer counterparts about the requirements engineering and software development processes.

Creativity. The analyst is not merely a scribe who records whatever customers say. The best analysts invent requirements (Robertson 2002). They conceive innovative product capabilities, imagine new markets and business opportunities, and think of ways to surprise and delight their customers. A really valuable analyst finds creative ways to satisfy needs that users didn't even know they had.

Essential Analyst Knowledge

In addition to the specific capabilities and personal characteristics just described, requirements analysts need a breadth of knowledge, much of which is gained through experience. Start with a solid understanding of contemporary requirements engineering techniques and the ability to apply them in the context of various software development life cycles. The effective analyst has a rich tool kit of techniques available and knows when—and when not—to use each one.

Analysts need to thread requirements development and management activities through the entire product life span. An analyst with a sound understanding of project management, risk management, and quality engineering can help prevent requirements issues from torpedoing the project. In a commercial software development setting, the analyst will benefit from knowledge of product management concepts and how enterprise software products are positioned and developed.

Application domain knowledge is a powerful asset for an effective analyst. The business-savvy analyst can minimize miscommunications with users. Analysts who understand the application domain often detect unstated assumptions and implicit requirements. They can suggest ways that users could improve their business processes. Such analysts sometimes propose valuable functionality that no user thought of. Conversely, they do a better job of detecting gold plating than does someone who's unfamiliar with the problem domain.

PREVIOUS NEXT
PREVIOUS NEXT

The Making of an Analyst

Great requirements analysts are grown, not trained. The job includes many "soft skills" that are more people-oriented than technological. There is no standard educational curriculum or job description for a requirements analyst. An organization's analysts will come from diverse backgrounds and they'll likely have gaps in their knowledge and skill sets. All analysts should decide which of the knowledge and skills described in this chapter pertain to their situation and actively seek the information that will let them do a first-rate job. Patricia Ferdinandi (2002) describes skill levels that junior, fully proficient, and lead requirements analysts should exhibit in various categories: practical experience, engineering, project management, techniques and tools, quality, and personality. All new analysts will benefit from mentoring and coaching from those who have more experience, perhaps in the form of an apprenticeship. Let's explore how people with different backgrounds might move into the analyst role.

The Former User

Many corporate IT departments have business analysts who migrated into that role following a career as a user. These individuals have a solid understanding of the business and the work environment, and they easily can gain the trust of their former colleagues. They speak the user's language, and they know the existing systems and business processes.

On the downside, former users who are now in the analyst role often know little about software engineering or how to communicate with technical people. If they aren't familiar with analysis modeling techniques, they will express all information in textual form. Users who become requirements analysts will need to learn more about the technical side of software development so that they can represent information in the most appropriate forms for their multiple audiences.

Some former users believe that their understanding of what is needed is better than that of the current users, so they don't solicit or respect input from those who will use the new system. Recent users can be stuck in the here-and-now of the current ways of working, such that they don't see opportunities to improve business processes with the help of a new information system. It's also easy for a former user to think of requirements strictly from a user interface perspective. Focusing on solution ideas can impose unnecessary design constraints from the outset and often fails to solve the real problem.

From Medical Technologist to Requirements Analyst

True Stories The senior manager of a medical devices division in a large company had a problem. "Two years ago, I hired three medical technologists into my division to represent our customers' needs," he said. "They've done a great job, but they are no longer current in medical technology, so they can't speak accurately for what our customers need today. But what's a reasonable career path for them now?"

This manager's former medical technologists might be candidates to become requirements analysts. Although they aren't up on the latest happenings in the hospital laboratory, they can still communicate with other med techs. Spending two years in a product development environment gave them a good appreciation for how it works. They might need some additional training in requirements-writing techniques, but these employees have accumulated a variety of valuable experiences that could make them effective analysts.

The Former Developer

Project managers who lack a dedicated requirements analyst often expect a developer to do the job.

Unfortunately, the skills and personality needed for requirements development aren't the same as those needed for software development. The stereotypical "geek" isn't the most socially adroit of human beings. A few developers have little patience with users, considering them a necessary evil to be dealt with quickly so that the developer can hustle back to the real job of cutting code. Of course, many developers recognize the criticality of the requirements process and are willing to work as analysts when necessary. Those who enjoy collaborating with customers to understand the needs that drive software development are good candidates to specialize in requirements analysis.

The developer-turned-analyst might need to learn more about the business domain. Developers can easily lapse into technical thinking and jargon, focusing on the software to be built instead of the users' needs. Developers will benefit from training and mentoring in the soft skills that the best analysts master, such as effective listening, negotiating, and facilitating.

The Subject Matter Expert

Ralph Young (2001) recommends having the requirements analyst be an application domain expert or a subject matter expert, as opposed to being a typical user: "SMEs can determine . . . whether the requirements are reasonable, how they extend the existing system, how the proposed architecture should be designed, and impacts on users, among other areas." Some product development organizations hire expert users of their products who have extensive domain experience into their companies to serve either as analysts or as user representatives.

The requirements analyst who is a domain expert might specify the system's requirements to suit his own preferences, rather than addressing the legitimate needs of the various user classes. SMEs sometimes target a high-end, all-inclusive system, when in fact a less comprehensive solution might meet most users' needs. It often works better to have a requirements analyst from the development team work with the SME, who then serves as a key user representative (product champion). (The product champion is described in Chapter 6.)

PREVIOUS NEXT
PREVIOUS NEXT

Creating a Collaborative Environment

Software projects sometimes experience strained relationships between analysts, developers, users, managers, and marketing. The parties don't always trust each other's motivations or appreciate each other's needs and constraints. In reality, though, the producers and consumers of a software product share common objectives. For corporate information systems development, all parties work for the same company so they all benefit from improvements to the corporate bottom line. For commercial products, happy customers generate revenue for the producer and satisfaction for the developers.The requirements analyst has the major responsibility for forging a collaborative relationship with the user representatives and other project stakeholders. An effective analyst appreciates the challenges that both business and technical stakeholders face and demonstrates respect for his or her collaborators at all times. The analyst steers the project participants toward a requirements agreement that leads to a win/win/win outcome in the following ways:

- Customers are delighted with the product.
- The developing organization is happy with the business outcomes.
- The development team members are proud of the good work they did on a challenging and rewarding project.

Next Steps

- If you are a requirements analyst, compare your own skills and knowledge with those described in this chapter. If you see gaps, select two specific areas for improvement and begin closing those gaps immediately by reading, practicing, finding a mentor, or taking a class.

- Select one new requirements engineering practice from this book to learn more about and try to apply it starting next week—literally! Select two or three additional practices to begin applying within a month. Choose others as long-term improvements, five or six months from now. Identify the situation to which you wish to apply each new practice, the benefits that you hope it will provide, and any help or additional information you might need. Think about whose cooperation you'll need to use the new techniques. Identify any barriers that might impede your ability to use the practice and consider who could help you break down those barriers.

PREVIOUS NEXT