Microsoft

Software Requirements

Third Edition

Karl Wiegers and Joy Beatty

Sample Chapters

To learn more about this book visit:
http://aka.ms/SoftwareReq3E/details

Contents

CHAPTER 6

Finding the voice of the user

Jeremy walked into the office of Ruth Gilbert, the director of the Drug Discovery Division at Contoso Pharmaceuticals. Ruth had asked the information technology team that supported Contoso's research organization to build a new application to help the research chemists accelerate their exploration for new drugs. Jeremy was assigned as the business analyst for the project. After introducing himself and discussing the project in broad terms, Jeremy said to Ruth, "I'd like to talk with some of your chemists to understand their requirements for the system. Who might be some good people to start with?"

Ruth replied, "I did that same job for five years before I became the division director three years ago. You don't really need to talk to any of my people; I can tell you everything you need to know about this project."

Jeremy was concerned. Scientific knowledge and technologies change quickly, so he wasn't sure if Ruth could adequately represent the current and future needs for users of this complex application. Perhaps there were some internal politics going on that weren't apparent and there was a good reason for Ruth to create a buffer between Jeremy and the actual users. After some discussion, though, it became clear that Ruth didn't want any of her people involved directly with the project.

"Okay," Jeremy agreed reluctantly. "Maybe I can start by doing some document analysis and bring questions I have to you. Can we set up a series of interviews for the next couple of weeks so I can understand the kinds of things you expect your scientists to be able to do with this new system?"

"Sorry, I'm swamped right now," Ruth told him. "I can give you a couple of hours in about three weeks to clarify things you're unsure about. Just go ahead and start writing the requirements. When we meet, then you can ask me any questions you still have. I hope that will let you get the ball rolling on this project."

If you share our conviction that customer involvement is a critical factor in delivering excellent software, you will ensure that the business analyst (BA) and project manager for your project will work hard to engage appropriate customer representatives from the outset. Success in software requirements, and hence in software development, depends on getting the voice of the user close to the ear of the developer. To find the voice of the user, take the following steps:

- Identify the different classes of users for your product.
- Select and work with individuals who represent each user class and other stakeholder groups.
- Agree on who the requirements decision makers are for your project.

Customer involvement is the best way to avoid the expectation gap described in Chapter 2, "Requirements from the customer's perspective," a mismatch between the product that customers expect to receive and what developers build. It's not enough simply to ask a few customers or their manager what they want once or twice and then start coding. If developers build exactly what customers initially request, they'll probably have to build it again because customers often don't know what they really need. In addition, the BAs might not be talking to the right people or asking the right questions.

The features that users present as their "wants" don't necessarily equate to the functionality they need to perform their tasks with the new product. To gain a more accurate view of user needs, the business analyst must collect a wide range of user input, analyze and clarify it, and specify just what needs to be built to let users do their jobs. The BA has the lead responsibility for recording the new system's necessary capabilities and properties and for communicating that information to other stakeholders. This is an iterative process that takes time. If you don't invest the time to achieve this shared understanding—this common vision of the intended product—the certain outcomes are rework, missed deadlines, cost overruns, and customer dissatisfaction.

User classes

People often talk about "the user" for a software system as though all users belong to a monolithic group with similar characteristics and needs. In reality, most products of any size appeal to a diversity of users with different expectations and goals. Rather than thinking of "the user" in singular, spend some time identifying the multiple user classes and their roles and privileges for your product.

Classifying users

Chapter 2 described many of the types of stakeholders that a project might have. As shown in Figure 6-1, a user class is a subset of the product's users, which is a subset of the product's customers, which is a subset of its stakeholders. An individual can belong to multiple user classes. For example, an application's administrator might also interact with it as an ordinary user at times. A product's users might differ—among other ways—in the following respects, and you can group users into a number of distinct *user classes* based on these sorts of differences:

- Their access privilege or security levels (such as ordinary user, guest user, administrator)
- The tasks they perform during their business operations
- The features they use
- The frequency with which they use the product
- Their application domain experience and computer systems expertise
- The platforms they will be using (desktop PCs, laptop PCs, tablets, smartphones, specialized devices)

- Their native language
- Whether they will interact with the system directly or indirectly

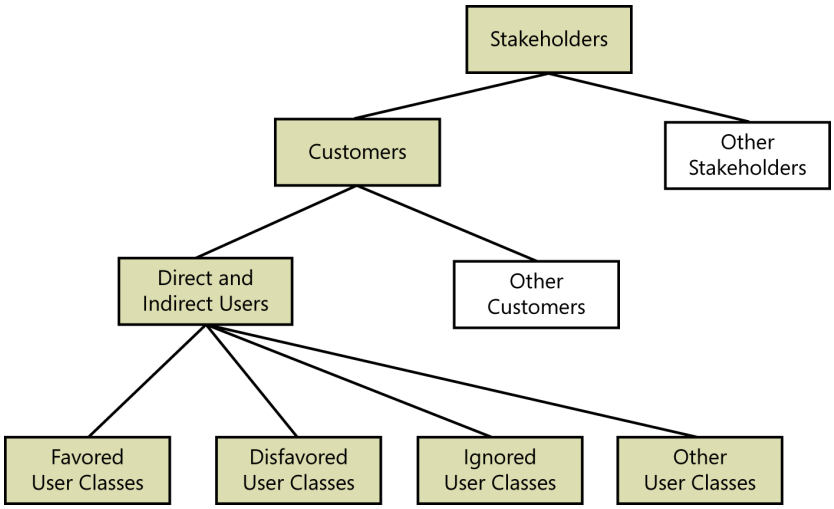

FIGURE 6-1 A hierarchy of stakeholders, customers, users, and user classes.

It's tempting to group users into classes based on their geographical location or the kind of company they work in. One company that creates software used in the banking industry initially considered distinguishing users based on whether they worked in a large commercial bank, a small commercial bank, a savings and loan institution, or a credit union. These distinctions really represent different market segments, though, not different user classes.

A better way to identify user classes is to think about the tasks that various users will perform with the system. All of those types of financial institutions will have tellers, employees who process loan applications, business bankers, and so forth. The individuals who perform such activities—whether they are job titles or simply roles—will have similar functional needs for the system across all of the financial institutions. Tellers all have to do more or less the same things, business bankers do more or less the same things, and so on. More logical user class names for a banking system therefore might include teller, loan officer, business banker, and branch manager. You might discover additional user classes by thinking of possible use cases, user stories, and process flows and who might perform them.

Certain user classes could be more important than others for a specific project. Favored user classes are those whose satisfaction is most closely aligned with achieving the project's business objectives. When resolving conflicts between requirements from different user classes or making priority decisions, favored user classes receive preferential treatment. This doesn't mean that the customers who are paying for the system (who might not be users at all) or those who have the most political clout should necessarily be favored. It's a matter of alignment with the business objectives.

Disfavored user classes are groups who aren't supposed to use the product for legal, security, or safety reasons (Gause and Lawrence 1999). You might build in features to deliberately make it hard for disfavored users to do things they aren't supposed to do. Examples include access security

mechanisms, user privilege levels, antimalware features (for non-human users), and usage logging. Locking a user's account after four unsuccessful login attempts protects against access by the disfavored user class of "user impersonators," albeit at the risk of inconveniencing forgetful legitimate users. If my bank doesn't recognize the computer I'm using, it sends me an email message with a one-time access code I have to enter before I can log on. This feature was implemented because of the disfavored user class of "people who might have stolen my banking information."

You might elect to ignore still other user classes. Yes, they will use the product, but you don't specifically build it to suit them. If there are any other groups of users that are neither favored, disfavored, nor ignored, they are of equal importance in defining the product's requirements.

Each user class will have its own set of requirements for the tasks that members of the class must perform. There could be some overlap between the needs of different user classes. Tellers, business bankers, and loan officers all might have to check a bank customer's account balance, for instance. Different user classes also could have different quality expectations, such as usability, that will drive user interface design choices. New or occasional users are concerned with how easy the system is to learn. Such users like menus, graphical user interfaces, uncluttered screen displays, wizards, and help screens. As users gain experience with the system, they become more interested in efficiency. They now value keyboard shortcuts, customization options, toolbars, and scripting facilities.

> **Trap** Don't overlook indirect user classes. They won't use your application themselves, instead accessing its data or services through other applications or through reports. Your customer once removed is still your customer.

User classes need not be human beings. They could be software agents performing a service on behalf of a human user, such as bots. Software agents can scan networks for information about goods and services, assemble custom news feeds, process your incoming email, monitor physical systems and networks for problems or intrusions, or perform data mining. Internet agents that probe websites for vulnerabilities or to generate spam are a type of disfavored non-human user class. If you identify these sorts of disfavored user classes, you might specify certain requirements not to meet their needs but rather to thwart them. For instance, website tools such as CAPTCHA that validate whether a user is a human being attempt to block such disruptive access by "users" you want to keep out.

Remember, users are a subset of customers, which are a subset of stakeholders. You'll need to consider a much broader range of potential sources of requirements than just direct and indirect user classes. For instance, even though the development team members aren't end users of the system they're building, you need their input on internal quality attributes such as efficiency, modifiability, portability, and reusability, as described in Chapter 14, "Beyond functionality." One company found that every installation of their product was an expensive nightmare until they introduced an "installer" user class so they could focus on requirements such as the development of a customization architecture for their product. Look well beyond the obvious end users when you're trying to identify stakeholders whose requirements input is necessary.

Identifying your user classes

Identify and characterize the different user classes for your product early in the project so you can elicit requirements from representatives of each important class. A useful technique for this is a collaboration pattern developed by Ellen Gottesdiener called "expand then contract" (Gottesdiener 2002). Start by asking the project sponsor who he expects to use the system. Then brainstorm as many user classes as you can think of. Don't get nervous if there are dozens at this stage; you'll condense and categorize them later. It's important not to overlook a user class, which can lead to problems later when someone complains that the delivered solution doesn't meet her needs. Next, look for groups with similar needs that you can either combine or treat as a major user class with several subclasses. Try to pare the list down to about 15 or fewer distinct user classes.

One company that developed a specialized product for about 65 corporate customers initially regarded each company as a distinct user with unique needs. Grouping their customers into just six user classes greatly simplified their requirements challenges. Donald Gause and Gerald Weinberg (1989) offer much advice about casting a wide net to identify potential users, pruning the user list, and seeking specific users to participate in the project.

Various analysis models can help you identify user classes. The external entities shown outside your system on a context diagram (see Chapter 5, "Establishing the business requirements") are candidates for user classes. A corporate organization chart can also help you discover potential users and other stakeholders (Beatty and Chen 2012). Figure 6-2 illustrates a portion of the organization chart for Contoso Pharmaceuticals. Nearly all of the potential users for the system are likely to be found somewhere in this chart. While performing stakeholder and user analysis, study the organization chart to look for:

- Departments that participate in the business process.
- Departments that are affected by the business process.
- Departments or role names in which either direct or indirect users might be found.
- User classes that span multiple departments.
- Departments that might have an interface to external stakeholders outside the company.

Organization chart analysis reduces the likelihood that you will overlook an important class of users within that organization. It shows you where to seek potential representatives for specific user classes, as well as helping determine who the key requirements decision makers might be. You might find multiple user classes with diverse needs within a single department. Conversely, recognizing the same user class in multiple departments can simplify requirements elicitation. Studying the organization chart helps you judge how many user representatives you'll need to work with to feel confident that you thoroughly understand the broad user community's needs. Also try to understand what type of information the users from each department might supply based on their role in the organization and their department's perspective on the project.

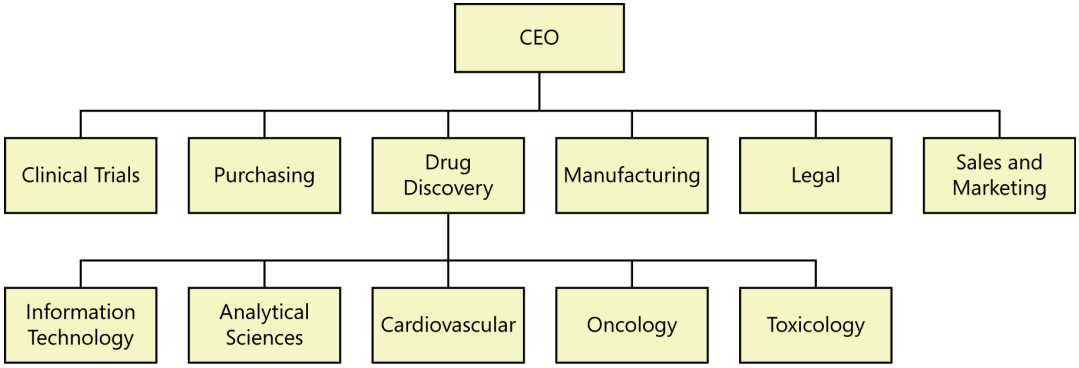

FIGURE 6-2 A portion of the organization chart for Contoso Pharmaceuticals.

Document the user classes and their characteristics, responsibilities, and physical locations in the software requirements specification (SRS) or in a requirements plan for your project. Check that information against any information you might already have about stakeholder profiles in the vision and scope document to avoid conflicts and duplication. Include all pertinent information you have about each user class, such as its relative or absolute size and which classes are favored. This will help the team prioritize change requests and conduct impact assessments later on. Estimates of the volume and type of system transactions help the testers develop a usage profile for the system so that they can plan their verification activities. The project manager and business analyst of the Chemical Tracking System discussed in earlier chapters identified the user classes and characteristics shown in Table 6-1.

TABLE 6-1 User classes for the Chemical Tracking System

Name	Number	Description
Chemists (favored)	Approximately 1,000 located in 6 buildings	Chemists will request chemicals from vendors and from the chemical stockroom. Each chemist will use the system several times per day, mainly for requesting chemicals and tracking chemical containers into and out of the laboratory. The chemists need to search vendor catalogs for specific chemical structures imported from the tools they use for drawing structures.
Buyers	5	Buyers in the purchasing department process chemical requests. They place and track orders with external vendors. They know little about chemistry and need simple query facilities to search vendor catalogs. Buyers will not use the system's container-tracking features. Each buyer will use the system an average of 25 times per day.
Chemical stockroom staff	6 technicians, 1 supervisor	The chemical stockroom staff manages an inventory of more than 500,000 chemical containers. They will supply containers from three stockrooms, request new chemicals from vendors, and track the movement of all containers into and out of the stockrooms. They are the only users of the inventory-reporting feature. Because of their high transaction volume, features that are used only by the chemical stockroom staff must be automated and efficient.
Health and Safety Department staff (favored)	1 manager	The Health and Safety Department staff will use the system only to generate predefined quarterly reports that comply with federal and state chemical usage and disposal reporting regulations. The Health and Safety Department manager will request changes in the reports periodically as government regulations change. These report changes are of the highest priority, and implementation will be time critical.

Consider building a catalog of user classes that recur across multiple applications. Defining user classes at the enterprise level lets you reuse those user class descriptions in future projects. The next system you build might serve the needs of some new user classes, but it probably will also be used by user classes from your earlier systems. If you do include the user-class descriptions in the project's SRS, you can incorporate entries from the reusable user-class catalog by reference and just write descriptions of any new groups that are specific to that application.

User personas

To help bring your user classes to life, consider creating a *persona* for each one, a description of a representative member of the user class (Cooper 2004; Leffingwell 2011). A persona is a description of a hypothetical, generic person who serves as a stand-in for a group of users having similar characteristics and needs. You can use personas to help you understand the requirements and to design the user experience to best meet the needs of specific user communities.

A persona can serve as a placeholder when the BA doesn't have an actual user representative at hand. Rather than having progress come to a halt, the BA can envision a persona performing a particular task or try to assess what the persona's preferences would be, thereby drafting a requirements starting point to be confirmed when an actual user is available. Persona details for a commercial customer include social and demographic characteristics and behaviors, preferences, annoyances, and similar information. Make sure the personas you create truly are representative of their user class, based on market, demographic, and ethnographic research.

Here's an example of a persona for one user class on the Chemical Tracking System:

> *Fred, 41, has been a chemist at Contoso Pharmaceuticals since he received his Ph.D. 14 years ago. He doesn't have much patience with computers. Fred usually works on two projects at a time in related chemical areas. His lab contains approximately 300 bottles of chemicals and gas cylinders. On an average day, he'll need four new chemicals from the stockroom. Two of these will be commercial chemicals in stock, one will need to be ordered, and one will come from the supply of proprietary Contoso chemical samples. On occasion, Fred will need a hazardous chemical that requires special training for safe handling. When he buys a chemical for the first time, Fred wants the material safety data sheet emailed to him automatically. Each year, Fred will synthesize about 20 new proprietary chemicals to go into the stockroom. Fred wants a report of his chemical usage for the previous month to be generated automatically and sent to him by email so that he can monitor his chemical exposure.*

As the business analyst explores the chemists' requirements, he can think about Fred as the archetype of this user class and ask himself, "What would Fred need to do?" Working with a persona makes the requirements thought process more tangible than if you simply contemplate what a whole faceless group of people might want. Some people choose a random human face of the appropriate gender to make a persona seem even more real.

Dean Leffingwell (2011) suggests that you design the system to make it easy for the individual described in your persona to use the application. That is, you focus on meeting that one (imaginary) person's needs. Provided you've created a persona that accurately represents the user class, this should help you do a good job of satisfying the needs and expectations of the whole class. As one colleague related, "On a project for servicing coin-operated vending machines, I introduced Dolly the Serviceperson and Ralph the Warehouse Supervisor. We wrote scenarios for them and they became part of the project team—virtually."

Connecting with user representatives

Every kind of project—corporate information systems, commercial applications, embedded systems, websites, contracted software—needs suitable representatives to provide the voice of the user. These users should be involved throughout the development life cycle, not just in an isolated requirements phase at the beginning of the project. Each user class needs someone to speak for it.

It's easiest to gain access to actual users when you're developing applications for deployment within your own company. If you're developing commercial software, you might engage people from your beta-testing or early-release sites to provide requirements input much earlier in the development process. (See the "External product champions" section later in this chapter). Consider setting up focus groups of current users of your products or your competitors' products. Instead of just guessing at what your users might want, ask some of them.

One company asked a focus group to perform certain tasks with various digital cameras and computers. The results indicated that the company's camera software took too long to perform the most common operation because of a design decision that was made to accommodate less likely scenarios as well. The company changed their next camera to reduce customer complaints about speed.

Be sure that the focus group represents the kinds of users whose needs should drive your product development. Include both expert and less experienced customers. If your focus group represents only early adopters or blue-sky thinkers, you might end up with many sophisticated and technically difficult requirements that few customers find useful.

Figure 6-3 illustrates some typical communication pathways that connect the voice of the user to the ear of the developer. One study indicated that employing more kinds of communication links and more direct links between developers and users led to more successful projects (Keil and Carmel 1995). The most direct communication occurs when developers can talk to appropriate users themselves, which means that the developer is also performing the business analyst role. This can work on very small projects, provided the developer involved has the appropriate BA skills, but it doesn't scale up to large projects with thousands of potential users and dozens of developers.

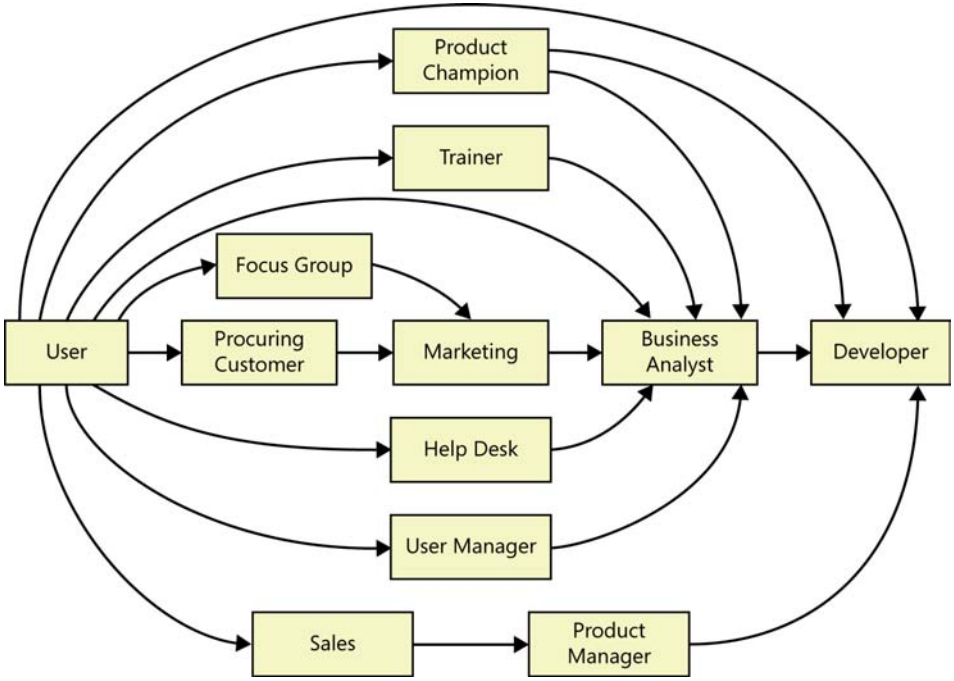

FIGURE 6-3 Some possible communication pathways between the user and the developer.

As in the children's game "Telephone," intervening layers between the user and the developer increase the chance of miscommunication and delay transmission. Some of these intervening layers add value, though, as when a skilled BA works with users or other participants to collect, evaluate, refine, and organize their input. Recognize the risks that you assume by using marketing staff, product managers, subject matter experts, or others as surrogates for the actual voice of the user. Despite the obstacles to—and the cost of—optimizing user representation, your product and your customers will suffer if you don't talk to the people who can provide the best information.

The product champion

Many years ago I worked in a small software development group that supported the scientific research activities at a major corporation. Each of our projects included a few key members of our user community to provide the requirements. We called these people *product champions* (Wiegers 1996). The product champion approach provides an effective way to structure that all-important customer-development collaborative partnership discussed in Chapter 2.

Each product champion serves as the primary interface between members of a single user class and the project's business analyst. Ideally, the champions will be actual users, not surrogates such as funding sponsors, marketing staff, user managers, or software developers imagining themselves to be users. Product champions gather requirements from other members of the user classes they represent and reconcile inconsistencies. Requirements development is thus a shared responsibility of the BA and selected users, although the BA should actually write the requirements documents. It's hard enough to write good requirements if you do it for a living; it is not realistic to expect users who have never written requirements before to do a good job.

The best product champions have a clear vision of the new system. They're enthusiastic because they see how it will benefit them and their peers. Champions should be effective communicators who are respected by their colleagues. They need a thorough understanding of the application domain and the solution's operating environment. Great product champions are in demand for other assignments, so you'll have to build a persuasive case for why particular individuals are critical to project success. For example, product champions can lead adoption of the application by the user community, which might be a success metric that managers will appreciate. We have found that good product champions made a huge difference in our projects, so we offer them public reward and recognition for their contributions.

Our software development teams enjoyed an additional benefit from the product champion approach. On several projects, we had excellent champions who spoke out on our behalf with their colleagues when the customers wondered why the software wasn't done yet. "Don't worry about it," the champions told their peers and their managers. "I understand and agree with the software team's approach to software engineering. The time we're spending on requirements will help us get the system we really need and will save time in the long run." Such collaboration helps break down the tension that can arise between customers and development teams.

The product champion approach works best if each champion is fully empowered to make binding decisions on behalf of the user class he represents. If a champion's decisions are routinely overruled by others, his time and goodwill are being wasted. However, the champions must remember that they are not the sole customers. Problems arise when the individual filling this critical liaison role doesn't adequately communicate with his peers and presents only his own wishes and ideas.

External product champions

When developing commercial software, it can be difficult to find product champions from outside your company. Companies that develop commercial products sometimes rely on internal subject matter experts or outside consultants to serve as surrogates for actual users, who might be unknown or difficult to engage. If you have a close working relationship with some major corporate customers, they might welcome the opportunity to participate in requirements elicitation. You might give external product champions economic incentives for their participation. Consider offering them discounts on the product or paying for the time they spend working with you on requirements. You still face the challenge of how to avoid hearing only the champions' requirements and overlooking the needs of other stakeholders. If you have a diverse customer base, first identify core requirements that are common to all customers. Then define additional requirements that are specific to individual corporate customers, market segments, or user classes.

Another alternative is to hire a suitable product champion who has the right background. One company that developed a retail point-of-sale and back-office system for a particular industry hired three store managers to serve as full-time product champions. As another example, my longtime family doctor, Art, left his medical practice to become the voice-of-the-physician at a medical software company. Art's new employer believed that it was worth the expense to hire a doctor to help the company build software that other doctors would accept. A third company hired several former employees from one of their major customers. These people provided valuable domain expertise as well as insight into the politics of the customer organization. To illustrate an alternative engagement model, one company had several corporate customers that used their invoicing systems extensively. Rather than bringing in product champions from the customers, the developing company sent BAs to the customer sites. Customers willingly dedicated some of their staff time to helping the BAs get the right requirements for the new invoicing system.

Anytime the product champion is a former or simulated user, watch out for disconnects between the champion's perceptions and the current needs of real users. Some domains change rapidly, whereas others are more stable. Regardless, if people aren't operating in the role anymore, they simply might have forgotten the intricacies of the daily job. The essential question is whether the product champion, no matter what her background or current job, can accurately represent the needs of today's real users.

Product champion expectations

To help the product champions succeed, document what you expect your champions to do. These written expectations can help you build a case for specific individuals to fill this critical role. Table 6-2 identifies some activities that product champions might perform (Wiegers 1996). Not every champion will do all of these; use this table as a starting point to negotiate each champion's responsibilities.

TABLE 6-2 Possible product champion activities

Category	Activities
Planning	■ Refine the scope and limitations of the product. ■ Identify other systems with which to interact. ■ Evaluate the impact of the new system on business operations. ■ Define a transition path from current applications or manual operations. ■ Identify relevant standards and certification requirements.
Requirements	■ Collect input on requirements from other users. ■ Develop usage scenarios, use cases, and user stories. ■ Resolve conflicts between proposed requirements within the user class. ■ Define implementation priorities. ■ Provide input regarding performance and other quality requirements. ■ Evaluate prototypes. ■ Work with other decision makers to resolve conflicts among requirements from different stakeholders. ■ Provide specialized algorithms.

Category	Activities
Validation and verification	■ Review requirements specifications. ■ Define acceptance criteria. ■ Develop user acceptance tests from usage scenarios. ■ Provide test data sets from the business. ■ Perform beta testing or user acceptance testing.
User aids	■ Write portions of user documentation and help text. ■ Contribute to training materials or tutorials. ■ Demonstrate the system to peers.
Change management	■ Evaluate and prioritize defect corrections and enhancement requests. ■ Dynamically adjust the scope of future releases or iterations. ■ Evaluate the impact of proposed changes on users and business processes. ■ Participate in making change decisions.

Multiple product champions

One person can rarely describe the needs for all users of an application. The Chemical Tracking System had four major user classes, so it needed four product champions selected from the internal user community at Contoso Pharmaceuticals. Figure 6-4 illustrates how the project manager set up a team of BAs and product champions to elicit the right requirements from the right sources. These champions were not assigned full time, but each one spent several hours per week working on the project. Three BAs worked with the four product champions to elicit, analyze, and document their requirements. (One BA worked with two product champions because the Buyer and the Health and Safety Department user classes were small and had few requirements.) One of the BAs assembled all the input into a unified SRS.

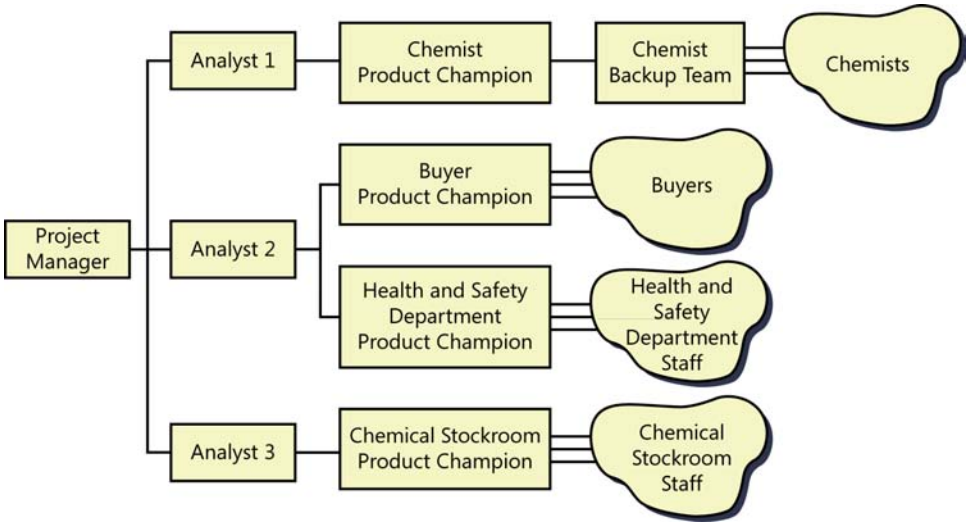

FIGURE 6-4 Product champion model for the Chemical Tracking System.

We didn't expect a single person to provide all the diverse requirements for the hundreds of chemists at Contoso. Don, the product champion for the Chemist user class, assembled a backup

team of five chemists from other parts of the company. They represented subclasses within the broad Chemist user class. This hierarchical approach engaged additional users in requirements development while avoiding the expense of massive workshops or dozens of individual interviews. Don always strove for consensus. However, he willingly made the necessary decisions when agreement wasn't achieved so the project could move ahead. No backup team was necessary when the user class was small enough or cohesive enough that one individual truly could represent the group's needs.[1]

The voiceless user class

A business analyst at Humongous Insurance was delighted that an influential user, Rebecca, agreed to serve as product champion for the new claims processing system. Rebecca had many ideas about the system features and user interface design. Thrilled to have the guidance of an expert, the development team happily complied with her requests. After delivery, though, they were shocked to receive many complaints about how hard the system was to use.

Rebecca was a power user. She specified usability requirements that were great for experts, but the 90 percent of users who *weren't* experts found the system unintuitive and difficult to learn. The BA didn't recognize that the claims processing system had at least two user classes. The large group of non–power users was disenfranchised in the requirements and user interface design processes. Humongous paid the price in an expensive redesign. The BA should have engaged at least one more product champion to represent the large class of nonexpert users.

Selling the product champion idea

Expect to encounter resistance when you propose the idea of having product champions on your projects. "The users are too busy." "Management wants to make the decisions." "They'll slow us down." "We can't afford it." "They'll run amok and scope will explode." "I don't know what I'm supposed to do as a product champion." Some users won't want to cooperate on a project that will make them change how they work or might even threaten their jobs. Managers are sometimes reluctant to delegate authority for requirements to ordinary users.

Separating business requirements from user requirements alleviates some of these discomforts. As an actual user, the product champion makes decisions at the user requirements level within the scope boundaries imposed by the business requirements. The management sponsor retains the authority to make decisions that affect the product vision, project scope, business-related priorities, schedule, or budget. Documenting and negotiating each product champion's role and responsibilities give candidate champions a comfort level about what they're being asked to do. Remind management that a product champion is a key contributor who can help the project achieve its business objectives.

[1] There's an interesting coda to this story. Years after I worked on this project, a man in a class I was teaching said he had worked at the company that Contoso Pharmaceuticals had contracted to build the Chemical Tracking System. The developers found that the requirements specification we created using this product champion model provided a solid foundation for the development work. The system was delivered successfully and was used at Contoso for many years.

If you encounter resistance, point out that insufficient user involvement is a leading cause of software project failure. Remind the protesters of problems they've experienced on previous projects that trace back to inadequate user input. Every organization has horror stories of new systems that didn't satisfy user needs or failed to meet unstated usability or performance expectations. You can't afford to rebuild or discard systems that don't measure up because no one understood the requirements. Product champions provide one way to get that all-important customer input in a timely way, not at the end of the project when customers are disappointed and developers are tired.

Product champion traps to avoid

The product champion model has succeeded in many environments. It works only when the product champions understand and sign up for their responsibilities, have the authority to make decisions at the user requirements level, and have time available to do the job. Watch out for the following potential problems:

- Managers override the decisions that a qualified and duly authorized product champion makes. Perhaps a manager has a wild new idea at the last minute, or thinks he knows what the users need. This behavior often results in dissatisfied users and frustrated product champions who feel that management doesn't trust them.
- A product champion who forgets that he is representing other customers and presents only his own requirements won't do a good job. He might be happy with the outcome, but others likely won't be.
- A product champion who lacks a clear vision of the new system might defer decisions to the BA. If all of the BA's ideas are fine with the champion, the champion isn't providing much help.
- A senior user might nominate a less experienced user as champion because she doesn't have time to do the job herself. This can lead to backseat driving from the senior user who still wishes to strongly influence the project's direction.

Beware of users who purport to speak for a user class to which they do not belong. Rarely, an individual might actively try to block the BA from working with the ideal contacts for some reason. On the Chemical Tracking System, the product champion for the chemical stockroom staff—herself a former chemist—initially insisted on providing what she thought were the needs of the chemist user class. Unfortunately, her input about current chemist needs wasn't accurate. It was difficult to convince her that this wasn't her job, but the BA didn't let her intimidate him. The project manager lined up a separate product champion for the chemists, who did a great job of collecting, evaluating, and relaying that community's requirements.

User representation on agile projects

Frequent conversations between project team members and appropriate customers are the most effective way to resolve many requirements issues and to flesh out requirements specifics when they are needed. Written documentation, however detailed, is an incomplete substitute for these ongoing communications. A fundamental tenet of Extreme Programming, one of the early agile development methods, is the presence of a full-time, on-site customer for these discussions (Jeffries, Anderson, and Hendrickson, 2001).

Some agile development methods include a single representative of stakeholders called a *product owner* in the team to serve as the voice of the customer (Schwaber 2004; Cohn 2010; Leffingwell 2011). The product owner defines the product's vision and is responsible for developing and prioritizing the contents of the product backlog. (The *backlog* is the prioritized list of user stories—requirements—for the product and their allocation to upcoming iterations, called sprints in the agile development method called Scrum.) The product owner therefore spans all three levels of requirements: business, user, and functional. He essentially straddles the product champion and business analyst functions, representing the customer, defining product features, prioritizing them, and so forth. Ultimately, someone does have to make decisions about exactly what capabilities to deliver in the product and when. In Scrum, that's the product owner's responsibility.

The ideal state of having a single product owner isn't always practical. We know of one company that was implementing a package solution to run their insurance business. The organization was too big and complex to have one person who understood everything in enough detail to make all decisions about the implementation. Instead, the customers selected a product owner from each department to own the priorities for the functionality used by that department. The company's CIO served as the lead product owner. The CIO understood the entire product vision, so he could ensure that the departments were on track to deliver that vision. He had responsibility for decision making when there were conflicts between department-level product owners.

The premises of the on-site customer and close customer collaboration with developers that agile methods espouse certainly are sound. In fact, we feel strongly that *all* development projects warrant this emphasis on user involvement. As you have seen, though, all but the smallest projects have multiple user classes, as well as numerous additional stakeholders whose interests must be represented. In many cases it's not realistic to expect a single individual to be able to adequately understand and describe the needs of all relevant user classes, nor to make all the decisions associated with product definition. Particularly with internal corporate projects, it will generally work better to use a representative structure like the product champion model to ensure adequate user engagement.

The product owner and product champion schemes are not mutually exclusive. If the product owner is functioning in the role of a business analyst, rather than as a stakeholder representative himself, he could set up a structure with one or more product champions to see that the most appropriate sources provide input. Alternatively, the product owner could collaborate with one or more business analysts, who then work with stakeholders to understand their requirements. The product owner would then serve as the ultimate decision maker.

"On-sight" customer

I once wrote programs for a research scientist who sat about 10 feet from my desk. John could provide instantaneous answers to my questions, provide feedback on user interface designs, and clarify our informally written requirements. One day John moved to a new office, around the corner on the same floor of the same building, about 100 feet away. I perceived an immediate drop in my programming productivity because of the cycle time delay in getting John's input. I spent more time fixing problems because sometimes I went down the wrong path before I could get a course correction. There's no substitute for having the right customers continuously available to the developers both on-site and "on-sight." Beware, though, of too-frequent interruptions that make it hard for people to refocus their attention on their work. It can take up to 15 minutes to reimmerse yourself into the highly productive, focused state of mind called *flow* (DeMarco and Lister 1999).

An on-site customer doesn't guarantee the desired outcome. My colleague Chris, a project manager, established a development team environment with minimal physical barriers and engaged two product champions. Chris offered this report: "While the close proximity seems to work for the development team, the results with product champions have been mixed. One sat in our midst and still managed to avoid us all. The new champion does a fine job of interacting with the developers and has truly enabled the rapid development of software." There is no substitute for having the right people, in the right role, in the right place, with the right attitude.

Resolving conflicting requirements

Someone must resolve conflicting requirements from different user classes, reconcile inconsistencies, and arbitrate questions of scope that arise. The product champions or product owner can handle this in many, but likely not all, cases. Early in the project, determine who the decision makers will be for requirements issues, as discussed in Chapter 2. If it's not clear who is responsible for making these decisions or if the authorized individuals abdicate their responsibilities, the decisions will fall to the developers or analysts by default. Most of them don't have the necessary knowledge and perspective

to make the best business decisions, though. Analysts sometimes defer to the loudest voice they hear or to the person highest on the food chain. Though understandable, this is not the best strategy. Decisions should be made as low in the organization's hierarchy as possible by well-informed people who are close to the issues.

Table 6-3 identifies some requirements conflicts that can arise on projects and suggests ways to handle them. The project's leaders need to determine who will decide what to do when such situations arise, who will make the call if agreement is not reached, and to whom significant issues must be escalated when necessary.

TABLE 6-3 Suggestions for resolving requirements disputes

Disagreement between	How to resolve
Individual users	Product champion or product owner decides
User classes	Favored user class gets preference
Market segments	Segment with greatest impact on business success gets preference
Corporate customers	Business objectives dictate direction
Users and user managers	Product owner or product champion for the user class decides
Development and customers	Customers get preference, but in alignment with business objectives
Development and marketing	Marketing gets preference

Trap Don't justify doing whatever any customer demands because "The customer is always right." We all know the customer is *not* always right (Wiegers 2011). Sometimes, a customer is unreasonable, uninformed, or in a bad mood. The customer always has a point, though, and the software team must understand and respect that point.

These negotiations don't always turn out the way the analyst might hope. Certain customers might reject all attempts to consider reasonable alternatives and other points of view. We've seen cases where marketing never said no to a customer request, no matter how infeasible or expensive. The team needs to decide who will be making decisions on the project's requirements before they confront these types of issues. Otherwise, indecision and the revisiting of previous decisions can stall the project in endless wrangling. If you're a BA caught in this dilemma, rely on your organizational structure and processes to work through the disagreements. But, as we've cautioned before, there aren't any easy solutions if you're working with truly unreasonable people.

Next steps

- Relate Figure 6-3 to the way you hear the voice of the user in your own environment. Do you encounter any problems with your current communication links? Identify the shortest and most effective communication paths that you can use to elicit user requirements in the future.
- Identify the different user classes for your project. Which ones are favored? Which, if any, are disfavored? Who would make a good product champion for each important user class? Even if the project is already underway, the team likely would benefit from having product champions involved.
- Starting with Table 6-2, define the activities you would like your product champions to perform. Negotiate the specific contributions with each candidate product champion and his or her manager.
- Determine who the decision makers are for requirements issues on your project. How well does your current decision-making approach work? Where does it break down? Are the right people making decisions? If not, who should be doing it? Suggest processes that the decision makers should use for reaching agreement on requirements issues.

CHAPTER 21

Enhancement and replacement projects

Most of this book describes requirements development as though you are beginning a new software or system development project, sometimes called a *green-field project*. However, many organizations devote much of their effort to enhancing or replacing existing information systems or building new releases of established commercial products. Most of the practices described in this book are appropriate for enhancement and replacement projects. This chapter provides specific suggestions as to which practices are most relevant and how to use them.

An *enhancement project* is one in which new capabilities are added to an existing system. Enhancement projects might also involve correcting defects, adding new reports, and modifying functionality to comply with revised business rules or needs.

A *replacement* (or *reengineering*) *project* replaces an existing application with a new custom-built system, a commercial off-the-shelf (COTS) system, or a hybrid of those. Replacement projects are most commonly implemented to improve performance, cut costs (such as maintenance costs or license fees), take advantage of modern technologies, or meet regulatory requirements. If your replacement project will involve a COTS solution, the guidance presented in Chapter 22, "Packaged solution projects," will also be helpful.

Replacement and enhancement projects face some particular requirements issues. The original developers who held all the critical information in their heads might be long gone. It's tempting to claim that a small enhancement doesn't warrant writing any requirements. Developers might believe that they don't need detailed requirements if they are replacing an existing system's functionality. The approaches described in this chapter can help you to deal with the challenges of enhancing or replacing an existing system to improve its ability to meet the organization's current business needs.

The case of the missing spec

The requirements specification for the next release of a mature system often says, essentially, "The new system should do everything the old system does, except add these new features and fix those bugs." A business analyst once received just such a specification for version 5 of a major product. To find out exactly what the current release did, she looked at the SRS for version 4. Unfortunately, it also said, in essence, "Version 4 should do everything that version 3 does, except add these new features and fix those bugs." She followed the trail back, but every

SRS described just the differences that the new version should exhibit compared to the previous version. Nowhere was there a description of the original system. Consequently, everyone had a different understanding of the current system's capabilities. If you're in this situation, document the requirements for your project more thoroughly so that all the stakeholders—both present and future—understand what the system does.

Expected challenges

The presence of an existing system leads to common challenges that both enhancement and replacement projects will face, including the following:

- The changes made could degrade the performance to which users are accustomed.
- Little or no requirements documentation might be available for the existing system.
- Users who are familiar with how the system works today might not like the changes they are about to encounter.
- You might unknowingly break or omit functionality that is vital to some stakeholder group.
- Stakeholders might take this opportunity to request new functionality that seems like a good idea but isn't really needed to meet the business objectives.

Even if there is existing documentation, it might not prove useful. For enhancement projects, the documentation might not be up to date. If the documentation doesn't match the existing application's reality, it is of limited use. For replacement systems, you also need to be wary of carrying forward *all* of the requirements, because some of the old functionality probably should not be migrated.

One of the major issues in replacement projects is validating that the reasons for the replacement are sound. There need to be justifiable business objectives for the change. When existing systems are being completely replaced, organizational processes might also have to change, which makes it harder for people to accept a new system. The change in business processes, change in the software system, and learning curve of a new system can disrupt current operations.

Requirements techniques when there is an existing system

Table 21-1 describes the most important requirements development techniques to consider when working on enhancement and replacement projects.

TABLE 21-1 Valuable requirements techniques for enhancement and replacement projects

Technique	Why it's relevant
Create a feature tree to show changes	■ Show features being added. ■ Identify features from the existing system that won't be in the new system.
Identify user classes	■ Assess who is affected by the changes. ■ Identify new user classes whose needs must be met.
Understand business processes	■ Understand how the current system is intertwined with stakeholders' daily jobs and the impacts of it changing. ■ Define new business processes that might need to be created to align with new features or a replacement system.
Document business rules	■ Record business rules that are currently embedded in code. ■ Look for new business rules that need to be honored. ■ Redesign the system to better handle volatile business rules that were expensive to maintain.
Create use cases or user stories	■ Understand what users must be able to do with the system. ■ Understand how users expect new features to work. ■ Prioritize functionality for the new system.
Create a context diagram	■ Identify and document external entities. ■ Extend existing interfaces to support new features. ■ Identify current interfaces that might need to be changed.
Create an ecosystem map	■ Look for other affected systems. ■ Look for new, modified, and obsolete interfaces between systems.
Create a dialog map	■ See how new screens fit into the existing user interface. ■ Show how the workflow screen navigation will change.
Create data models	■ Verify that the existing data model is sufficient or extend it for new features. ■ Verify that all of the data entities and attributes are still needed. ■ Consider what data has to be migrated, converted, corrected, archived, or discarded.
Specify quality attributes	■ Ensure that the new system is designed to fulfill quality expectations. ■ Improve satisfaction of quality attributes over the existing system.
Create report tables	■ Convert existing reports that are still needed. ■ Define new reports that aren't in the old system.
Build prototypes	■ Engage users in the redevelopment process. ■ Prototype major enhancements if there are uncertainties.
Inspect requirements specifications	■ Identify broken links in the traceability chain. ■ Determine if any previous requirements are obsolete or unnecessary in the replacement system.

Enhancement projects provide an opportunity to try new requirements methods in a small-scale and low-risk way. The pressure to get the next release out might make you think that you don't have time to experiment with requirements techniques, but enhancement projects let you tackle the learning curve in bite-sized chunks. When the next big project comes along, you'll have some experience and confidence in better requirements practices.

Suppose that a customer requests that a new feature be added to a mature product. If you haven't worked with user stories before, explore the new feature from the user-story perspective, discussing with the requester the tasks that users will perform with that feature. Practicing on this project reduces the risk compared to applying user stories for the first time on a green-field project, when your skill might mean the difference between success and high-profile failure.

Prioritizing by using business objectives

Enhancement projects are undertaken to add new capabilities to an existing application. It's easy to get caught up in the excitement and start adding unnecessary capabilities. To combat this risk of gold-plating, trace requirements back to business objectives to ensure that the new features are needed and to select the highest-impact features to implement first. You also might need to prioritize enhancement requests against the correction of defects that had been reported against the old system.

Also be wary of letting unnecessary new functionality slip into replacement projects. The main focus of replacement projects is to migrate existing functionality. However, customers might imagine that if you are developing a new system anyway, it is easy to add lots of new capabilities right away. Many replacement projects have collapsed because of the weight of uncontrolled scope growth. You're usually better off building a stable first release and adding more features through subsequent enhancement projects, provided the first release allows users to do their jobs.

Replacement projects often originate when stakeholders want to add functionality to an existing system that is too inflexible to support the growth or has technology limitations. However, there needs to be a clear business objective to justify implementing an expensive new system (Devine 2008). Use the anticipated cost savings from a new system (such as through reduced maintenance of an old, clunky system) plus the value of the new desired functionality to justify a system replacement project.

Also look for existing functionality that doesn't need to be retained in a replacement system. Don't replicate the existing system's shortcomings or miss an opportunity to update a system to suit new business needs and processes. For example, the BA might ask users, "Do you use *<a particular menu option>*?" If you consistently hear "I never do that," then maybe it isn't needed in the replacement system. Look for usage data that shows what screens, functions, or data entities are rarely accessed in the current system. Even the existing functionality has to map to current and anticipated business objectives to warrant re-implementing it in the new system.

> **Trap** Don't let stakeholders get away with saying "I have it today, so I need it in the new system" as a default method of justifying requirements.

Mind the gap

A *gap analysis* is a comparison of functionality between an existing system and a desired new system. A gap analysis can be expressed in different ways, including use cases, user stories, or features. When enhancing an existing system, perform a gap analysis to make sure you understand why it isn't currently meeting your business objectives.

Gap analysis for a replacement project entails understanding existing functionality and discovering the desired new functionality (see Figure 21-1). Identify user requirements for the existing system that stakeholders want to have re-implemented in the new system. Also, elicit new user requirements that the existing system does not address. Consider any change requests that were never implemented

in the existing system. Prioritize the existing user requirements and the new ones together. Prioritize closing the gaps using business objectives as described in the previous section or the other prioritization techniques presented in Chapter 16, "First things first: Setting requirement priorities."

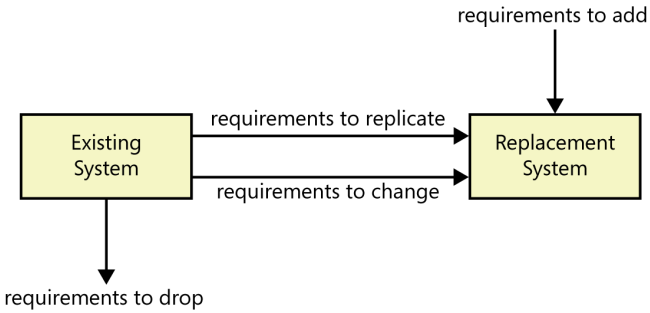

FIGURE 21-1 When you are replacing an existing system, some requirements will be implemented unchanged, some will be modified, some will be discarded, and some new requirements might be added.

Maintaining performance levels

Existing systems set user expectations for performance and throughput. Stakeholders almost always have key performance indicators (KPIs) for existing processes that they will want to maintain in the new system. A key performance indicator model (KPIM) can help you identify and specify these metrics for their corresponding business processes (Beatty and Chen 2012). The KPIM helps stakeholders see that even if the new system will be different, their business outcomes will be at least as good as before.

Unless you explicitly plan to maintain them, performance levels can be compromised as systems are enhanced. Stuffing new functionality into an existing system might slow it down. One data synchronization tool had a requirement to update a master data set from the day's transactions. It needed to run every 24 hours. In the initial release of the tool, the synchronization started at midnight and took about one hour to execute. After some enhancements to include additional attributes, merging, and synchronicity checks, the synchronization took 20 hours to execute. This was a problem, because users expected to have fully synchronized data from the night before available when they started their workday at 8:00 A.M. The maximum time to complete the synchronization was never explicitly specified, but the stakeholders assumed it could be done overnight in less than eight hours.

For replacement systems, prioritize the KPIs that are most important to maintain. Look for the business processes that trace to the most important KPIs and the requirements that enable those business processes; these are the requirements to implement first. For instance, if you're replacing a loan application system in which loan processors can enter 10 loans per day, it might be important to maintain at least that same throughput in the new system. The functionality that allows loan processers to enter loans should be some of the earliest implemented in the new system, so the loan processors can maintain their productivity.

When old requirements don't exist

Most older systems do not have documented—let alone accurate—requirements. In the absence of reliable documentation, teams might reverse-engineer an understanding of what the system does from the user interfaces, code, and database. We think of this as "software archaeology." To maximize the benefit from reverse engineering, the archaeology expedition should record what it learns in the form of requirements and design descriptions. Accumulating accurate information about certain portions of the current system positions the team to enhance a system with low risk, to replace a system without missing critical functionality, and to perform future enhancements efficiently. It halts the knowledge drain, so future maintainers better understand the changes that were just made.

If updating the requirements is overly burdensome, it will fall by the wayside as busy people rush on to the next change request. Obsolete requirements aren't helpful for future enhancements. There's a widespread fear in the software industry that writing documentation will consume too much time; the knee-jerk reaction is to neglect all opportunities to update requirements documentation. But what's the cost if you *don't* update the requirements and a future maintainer (perhaps you!) has to regenerate that information? The answer to this question will let you make a thoughtful business decision concerning whether to revise the requirements documentation when you change or re-create the software.

When the team performs additional enhancements and maintenance over time, it can extend these fractional knowledge representations, steadily improving the system documentation. The incremental cost of recording this newly found knowledge is small compared with the cost of someone having to rediscover it later on. Implementing enhancements almost always necessitates further requirements development, so add those new requirements to an existing requirements repository, if there is one. If you're replacing an old system, you have an opportunity to document the requirements for the new one and to keep the requirements up to date with what you learn throughout the project. Try to leave the requirements in better shape than you found them.

Which requirements should you specify?

It's not always worth taking the time to generate a complete set of requirements for an entire production system. Many options lie between the two extremes of continuing forever with no requirements documentation and reconstructing a perfect requirements set. Knowing why you'd like to have written requirements available lets you judge whether the cost of rebuilding all—or even part—of the specification is a sound investment.

Perhaps your current system is a shapeless mass of history and mystery like the one in Figure 21-2. Imagine that you've been asked to implement some new functionality in region A in this figure. Begin by recording the new requirements in a structured SRS or in a requirements management tool. When you add the new functionality, you'll have to figure out how it interfaces to or fits in with the existing system. The bridges in Figure 21-2 between region A and your current system represent these interfaces. This analysis provides insight into the white portion of the current system, region B. In addition to the requirements for region A, this insight is the new knowledge you need to capture.

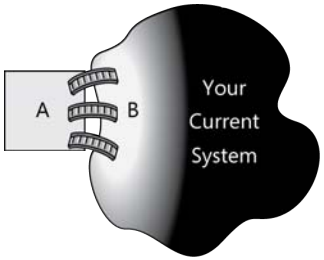

FIGURE 21-2 Adding enhancement A to an ill-documented existing system provides some visibility into the B area.

Rarely do you need to document the entire existing system. Focus detailed requirements efforts on the changes needed to meet the business objectives. If you're replacing a system, start by documenting the areas prioritized as most important to achieve the business objectives or those that pose the highest implementation risk. Any new requirements identified during the gap analysis will need to be specified at the same level of precision and using the same techniques as you would for a new system.

Level of detail

One of the biggest challenges is determining the appropriate level of detail at which to document requirements gleaned from the existing system. For enhancements, defining requirements for the new functionality alone might be sufficient. However, you will usually benefit from documenting all of the functionality that closely relates to the enhancement, to ensure that the change fits in seamlessly (region B in Figure 21-2). You might want to create business processes, user requirements, and/or functional requirements for those related areas. For example, let's say you are adding a discount code feature to an existing shopping cart function, but you don't have any documented requirements for the shopping cart. You might be tempted to write just a single user story: "As a customer, I need to be able to enter a discount code so I can get the cheapest price for the product." However, this user story alone lacks context, so consider capturing other user stories about shopping cart operations. That information could be valuable the next time you need to modify the shopping cart function.

I worked with one team that was just beginning to develop the requirements for version 2 of a major product with embedded software. They hadn't done a good job on the requirements for version 1, which was currently being implemented. The lead BA wondered, "Is it worth going back to improve the SRS for version 1?" The company anticipated that this product line would be a major revenue generator for at least 10 years. They also planned to reuse some of the core requirements in several spin-off products. In this case, it made sense to improve the requirements documentation for version 1 because it was the foundation for all subsequent development work in this product line. Had they been working on version 5.3 of a well-worn system that they expected to retire within a year, reconstructing a comprehensive set of requirements wouldn't have been a wise investment.

Trace Data

Requirements trace data for existing systems will help the enhancement developer determine which components she might have to modify because of a change in a specific requirement. In an ideal world, when you're replacing a system, the existing system would have a full set of functional requirements such that you could establish traceability between the old and new systems to avoid overlooking any requirements. However, a poorly documented old system won't have trace information available, and establishing rigorous traceability for both existing and new systems is time consuming.

As with any new development, it's a good practice to create a traceability matrix to link the new or changed requirements to the corresponding design elements, code, and test cases. Accumulating trace links as you perform the development work takes little effort, whereas it's a great deal of work to regenerate the links from a completed system. For replacement systems, perform requirements tracing at a high level: make a list of features and user stories for the existing system and prioritize to determine which of those will be implemented in the new system. See Chapter 29, "Links in the requirements chain," for more information on tracing requirements.

How to discover the requirements of an existing system

In enhancement and replacement projects, even if you don't have existing documentation, you do have a system to work from to discover the relevant requirements. During enhancement projects, consider drawing a dialog map for the new screens you have to add, showing the navigation connections to and from existing display elements. You might write use cases or user stories that span the new and existing functionality.

In replacement system projects, you need to understand all of the desired functionality, just as you do on any new development project. Study the user interface of the existing system to identify candidate functionality for the new system. Examine existing system interfaces to determine what data is exchanged between systems today. Understand how users use the current system. If no one understands the functionality and business rules behind the user interface, someone will need to look at the code or database to understand what's going on. Analyze any documentation that does exist—design documents, help screens, user manuals, training materials—to identify requirements.

You might not need to specify functional requirements for the existing system at all, instead creating models to fill the information void. Swimlane diagrams can describe how users do their jobs with the system today. Context diagrams, data flow diagrams, and entity-relationship diagrams are also useful. You might create user requirements, specifying them only at a high level without filling in all of the details. Another way to begin closing the information gap is to create data dictionary entries when you add new data elements to the system and modify existing definitions. The test suite might be useful as an initial source of information to recover the software requirements, because tests represent an alternative view of requirements.

Sometimes "good enough" is enough

A third-party assessment of current business analysis practices in one organization revealed that their teams did a fairly good job of writing requirements for new projects, but they failed to update the requirements as the products evolved through a series of enhancement releases. The BAs did create requirements for each enhancement project. However, they did not merge all of those revisions back into the requirements baseline. The organization's manager couldn't think of a measurable benefit from keeping the existing documentation 100 percent updated to reflect the implemented systems. He assumed that his requirements always reflected only 80 to 90 percent of the working software anyway, so there was little value in trying to perfect the requirements for an enhancement. This meant that future enhancement project teams would have to work with some uncertainty and close the gaps when needed, but that price was deemed acceptable.

Encouraging new system adoption

You're bound to run into resistance when changing or replacing an existing system. People are naturally reluctant to change. Introducing a new feature that will make users' jobs easier is a good thing. But users are accustomed to how the system works today, and you plan to modify that, which is not so good from the user's point of view. The issue is even bigger when you're replacing a system, because now you're changing more than just a bit of functionality. You're potentially changing the entire application's look and feel, its menus, the operating environment, and possibly the user's whole job. If you're a business analyst, project manager, or project sponsor, you have to anticipate the resistance and plan how you will overcome it, so the users will accept the new features or system.

An existing, established system is probably stable, fully integrated with surrounding systems, and well understood by users. A new system with all the same functionality might be none of these upon its initial release. Users might fear that the new system will disrupt their normal operations while they learn how to use it. Even worse, it might not support their current operations. Users might even be afraid of losing their jobs if the system automates tasks they perform manually today. It's not uncommon to hear users say that they will accept the new system only if it does everything the old system does—even if they don't personally use all of that functionality at present.

To mitigate the risk of user resistance, you first need to understand the business objectives and the user requirements. If either of these misses the mark, you will lose the users' trust quickly. During elicitation, focus on the benefits the new system or each feature will provide to the users. Help them understand the value of the proposed change to the organization as a whole. Keep in mind—even with enhancements—that just because something is new doesn't mean it will make the user's job easier. A poorly designed user interface can even make the system harder to use because the old features are harder to find, lost amidst a clutter of new options, or more cumbersome to access.

Our organization recently upgraded our document-repository tool to a new version to give us access to additional features and a more stable operating environment. During beta testing, I discovered that simple, common tasks such as checking out and downloading a file are now harder. In the previous version, you could check out a file in two clicks, but now it takes three or four, depending on the navigation path you choose. If our executive stakeholders thought these user interface changes were a big risk to user acceptance, they could invest in developing custom functionality to mimic the old system. Showing prototypes to users can help them get used to the new system or new features and reveal likely adoption issues early in the project.

One caveat with system replacements is that the key performance indicators for certain groups might be negatively affected, even if the system replacement provides a benefit for the organization as a whole. Let users know as soon as possible about features they might be losing or quality attributes that might degrade, so they can start to prepare for it. System adoption can involve as much emotion as logic, so expectation management is critical to lay the foundation for a successful rollout.

When you are migrating from an existing system, transition requirements are also important. Transition requirements describe the capabilities that the whole solution—not just the software application—must have to enable moving from the existing system to the new system (IIBA 2009). They can encompass data conversions, user training, organizational and business process changes, and the need to run both old and new systems in parallel for a period of time. Think about everything that will be required for stakeholders to comfortably and efficiently transition to the new way of working. Understanding transition requirements is part of assessing readiness and managing organizational change (IIBA 2009).

Can we iterate?

Enhancement projects are incremental by definition. Project teams can often adopt agile methods readily, by prioritizing enhancements using a product backlog as described in Chapter 20, "Agile projects." However, replacement projects do not always lend themselves to incremental delivery because you need a critical mass of functionality in the new application before users can begin using it to do their jobs. It's not practical for them to use the new system to do a small portion of their job and then have to go back to the old system to perform other functions. However, big-bang migrations are also challenging and unrealistic. It's difficult to replace in a single step an established system that has matured over many years and numerous releases.

One approach to implementing a replacement system incrementally is to identify functionality that can be isolated and begin by building just those pieces. We once helped a customer team to replace their current fulfillment system with a new custom-developed system. Inventory management represented about 10 percent of the total functionality of the entire fulfillment system. For the most part, the people who managed inventory were separate from the people who managed other parts of the fulfillment process. The initial strategy was to move just the inventory management

functionality to a new system of its own. This was ideal functionality to isolate for the first release because it affected just a subset of users, who then would primarily work only in the new system. The one downside side to the approach is that a new software interface had to be developed so that the new inventory system could pass data to and from the existing fulfillment system.

We had no requirements documentation for the existing system. But retaining the original system and turning off its inventory management piece provided a clear boundary for the requirements effort. We primarily wrote use cases and functional requirements for the new inventory system, based on the most important functions of the existing system. We created an entity-relationship diagram and a data dictionary. We drew a context diagram for the entire existing fulfillment system to understand integration points that might be relevant when we split inventory out of it. Then we created a new context diagram to show how inventory management would exist as an external system that interacts with the truncated fulfillment system.

Not all enhancement or replacement projects will be this clean. Most of them will struggle to overcome the two biggest challenges: a lack of documentation for the existing system, and a potential battle to get users to adopt the new system or features. However, using the techniques described in this chapter can help you actively mitigate these risks.