CHAPTER 10

Documenting the requirements

At the launch of a large project to build a commercial software company's next-generation flagship product, a senior manager convened about 60 employees in a daylong off-site "voice-of-the-customer workshop." These employees worked with facilitators to generate ideas for the new product. The manager compiled the results of these brainstorming sessions into a 100-page document. He called this a requirements specification, but in fact it was nothing more than a pile of information.

The information from the brain dump by all these smart people wasn't classified into various categories, organized logically, analyzed, or otherwise processed into anything that described a proposed software solution. Developers could not have gleaned what they needed to know about the new product from this massive collection of ideas. Certainly there were nuggets of valuable requirements buried among all the chaff. But simply collecting raw ideas and needs into a long list isn't an effective way to document and communicate software requirements.

Clear and effective communication is the core principle of requirements development—communication from people with needs to people who can conceive solutions, then to people who can implement and verify those solutions. A skilled business analyst will choose the most effective way to communicate each type of requirements information to each audience.

The result of requirements development is a documented agreement among stakeholders about the product to be built. As you saw in earlier chapters, the vision and scope document contains the business requirements, and user requirements can be captured in the form of use cases or user stories. The product's functional and nonfunctional requirements often are stored in a software requirements specification, or SRS, which is delivered to those who must design, build, and verify the solution. Recording requirements in an organized fashion that key project stakeholders can review helps ensure that they know what they're agreeing to.

This chapter addresses the purpose, structure, and contents of the SRS. We will describe the SRS as being a document, but it doesn't have to be in the form of a traditional word-processing document. In fact, documents pose numerous limitations:

- It's difficult to store descriptive attributes along with the requirements.
- Change management is clumsy.
- It's difficult to retain historical versions of the requirements.
- It's not easy to subset out a portion of requirements that are allocated to a particular iteration or keep track of those that were once approved but then deferred or canceled.

- It's hard to trace requirements to other development artifacts.
- Duplicating a requirement that logically fits in multiple places causes maintenance issues.

As alternatives, you might store information in a spreadsheet (which has many of the same limitations as a document), a Wiki, a database, or a requirements management (RM) tool (see Chapter 30, "Tools for requirements engineering"). Think of these as different possible repositories or containers for requirements information. No matter what form of requirements repository you use, you still need the same kinds of information. The SRS template described here is a helpful reminder of information to collect and how you might organize it.

Not everyone agrees that it's worth the time to document requirements. And on exploratory or highly volatile projects where you're not sure what solution you'll end up with, trying to keep up with changes in the requirements details adds little value. However, the cost of recording knowledge is small compared to the cost of acquiring that knowledge or regenerating it at some point in the future. The acts of specification and modeling help project participants think through and precisely state important things that a verbal discussion can leave ambiguous. If you are *100 percent certain* that no stakeholders will ever need a specific piece of information beyond the duration of their own short-term memories, then you don't need to record it. Otherwise, store it in some kind of a group memory.

You will never get perfect requirements. Remember that you are writing requirements for certain audiences. The amount of detail, the kinds of information you provide, and the way you organize it should all be intended to meet the needs of your audiences. Analysts quite naturally write requirements from their own point of view, but really they should write them to be most meaningful to those who have to understand the requirements and do work based on them. This is why it's important to have representatives of those audiences review the requirements to make sure they'll meet their needs.

Progressive refinement of detail is a key principle for effective requirements development. On most projects it's neither realistic nor necessary to pin down every requirement detail early in the project. Instead, think in terms of layers. You need to learn just enough about the requirements to be able to roughly prioritize them and allocate them to forthcoming releases or iterations. Then you can detail groups of requirements in a just-in-time fashion to give developers enough information so they can avoid excessive and unnecessary rework.

Don't expect even the finest requirements documentation to replace ongoing discussions throughout the project. Keep the communication lines open among the BA, development team, customer representatives, and other stakeholders so that they can quickly address the myriad issues that will arise.

Trap Do not rely on telepathy and clairvoyance as substitutes for solid requirements specification practices. They don't work, even though they seem to be the technical foundation for some software projects.

You can represent software requirements in several ways, including:

- Well-structured and carefully written natural language.
- Visual models that illustrate transformational processes, system states and changes between them, data relationships, logic flows, and the like.
- Formal specifications that define requirements by using mathematically precise specification languages.

Formal specifications provide the greatest rigor and precision, but few software developers—and even fewer customers—are familiar with them. Most projects don't demand this level of formality, but I'd certainly hope that the designers of high-risk systems like nuclear power plant control systems use formal specification methods. Structured natural language, augmented with visual models and other representation techniques (such as tables, mock-ups, photographs, and mathematical expressions), remains the most practical way for most software projects to document their requirements. The rest of this chapter addresses how you might organize the information in a software requirements specification. Chapter 11, "Writing excellent requirements," describes characteristics of high-quality requirements and offers many suggestions for how to write them.

The software requirements specification

The software requirements specification goes by many names in various organizations, although organizations do not use these terms in the same way. It is sometimes called a *business requirements document* (BRD), *functional specification*, *product specification*, *system specification*, or simply *requirements document*. Because "software requirements specification" is an industry-standard term, that's what we'll call it here (ISO/IEC/IEEE 2011).

The SRS states the functions and capabilities that a software system must provide, its characteristics, and the constraints that it must respect. It should describe as completely as necessary the system's behaviors under various conditions, as well as desired system qualities such as performance, security, and usability. The SRS is the basis for subsequent project planning, design, and coding, as well as the foundation for system testing and user documentation. However, it should not contain design, construction, testing, or project management details other than known design and implementation constraints. Even people working on agile projects need the kind of information found in a good SRS. They don't ordinarily collect all this information in a cohesive deliverable, but an SRS template provides a convenient reminder of what kinds of knowledge to explore. This chapter concludes with a section that describes how agile projects typically handle requirements specification.

Important A single requirements deliverable often cannot meet the needs of all audiences. Some people need to know just the business objectives, others want only a high-level big picture, still others want to see just the user's perspective, and yet others need all the details. This is one reason why we advocate creating the deliverables we call the vision and scope document, user requirements document, and software requirements specification. Don't expect all of your user representatives to read the detailed SRS, and don't expect developers to learn all they need from a set of use cases or user stories.

Numerous audiences rely on the SRS:

- Customers, the marketing department, and sales staff need to know what product they can expect to be delivered.
- Project managers base their estimates of schedule, effort, and resources on the requirements.
- Software development teams need to know what to build.
- Testers use it to develop requirements-based tests, test plans, and test procedures.
- Maintenance and support staff use it to understand what each part of the product is supposed to do.
- Documentation writers base user manuals and help screens on the SRS and the user interface design.
- Training personnel use the SRS and user documentation to develop educational materials.
- Legal staff ensures that the requirements comply with applicable laws and regulations.
- Subcontractors base their work on—and can be legally held to—the specified requirements.

If a desired capability or quality doesn't appear somewhere in the requirements agreement, no one should expect it to appear in the product.

How many specifications?

Most projects will create just one software requirements specification. This isn't practical for large projects, though. Large systems projects often write a system requirements specification, followed by separate software and perhaps hardware requirements specifications (ISO/IEC/IEEE 2011). One company was building a very complex process control application, with more than 100 people working for multiple years. This project had about 800 high-level requirements in its system requirements specification. The project was divided into 20 subprojects, each of which had its own software requirements specification with perhaps 800 or 900 requirements derived from the system requirements. This makes for a lot of documentation, but a large project becomes unmanageable if you don't take a divide-and-conquer approach.

At the other extreme, another company created just a single guiding document for each medium-sized project, which they called simply "The Spec." The Spec contained every piece of known information about the project: requirements, estimates, project plans, quality plans, test plans, tests, everything. Change management and version control on such an all-inclusive document is a nightmare. Nor is the information level in such an all-inclusive document suitable for each audience for requirements information.

A third company that began to adopt agile development practices stopped writing any formal documentation. Instead, they wrote user stories for a large project on sticky notes that they placed on their office walls. Unfortunately for one project, the adhesive on the sticky notes gradually failed. A couple of months into the project, it was normal for no-longer-sticky notes to flutter to the ground as someone walked by the wall.

Still another company took an intermediate approach. Although their projects weren't huge and could be specified in just 40 to 60 pages, some team members wanted to subdivide the SRS into as many as 12 separate documents: one SRS for a batch process, one for the reporting engine, and one for each of 10 reports. A document explosion like this causes headaches because it's hard to keep changes to them synchronized and to make sure the right people get all the information they need efficiently.

A better alternative for all of these situations is to store the requirements in a requirements management tool, as described in Chapter 30. An RM tool also helps greatly with the problem of whether to create a single SRS or multiple specifications for a project that plans multiple product releases or development iterations (Wiegers 2006). The SRS for any one portion of the product or for a given iteration then is just a report generated from the database contents based on certain query criteria.

You don't have to write the SRS for the entire product before beginning development, but you should capture the requirements for each increment before building that increment. Incremental development is appropriate when you want to get some functionality into the users' hands quickly. Feedback from using the early increments will shape the rest of the project. However, every project should baseline an agreement for each set of requirements before the team implements them. *Baselining* is the process of transitioning an SRS under development into one that has been reviewed and approved. Working from an agreed-upon set of requirements minimizes miscommunication and unnecessary rework. See Chapter 2, "Requirements from the customer's perspective," and Chapter 27, "Requirements management practices," for more about baselining.

It's important to organize and write the SRS so that the diverse stakeholders can understand it. Keep the following readability suggestions in mind:

- Use an appropriate template to organize all the necessary information.
- Label and style sections, subsections, and individual requirements consistently.

- Use visual emphasis (bold, underline, italics, color, and fonts) consistently and judiciously. Remember that color highlighting might not be visible to people with color blindness or when printed in grayscale.
- Create a table of contents to help readers find the information they need.
- Number all figures and tables, give them captions, and refer to them by number.
- If you are storing requirements in a document, define your word processor's cross-reference facility rather than hard-coded page or section numbers to refer to other locations within a document.
- If you are using documents, define hyperlinks to let the reader jump to related sections in the SRS or in other files.
- If you are storing requirements in a tool, use links to let the reader navigate to related information.
- Include visual representations of information when possible to facilitate understanding.
- Enlist a skilled editor to make sure the document is coherent and uses a consistent vocabulary and layout.

Labeling requirements

Every requirement needs a unique and persistent identifier. This allows you to refer to specific requirements in a change request, modification history, cross-reference, or requirements traceability matrix. It also enables reusing the requirements in multiple projects. Uniquely identified requirements facilitate collaboration between team members when they're discussing requirements, as in a peer review meeting. Simple numbered or bulleted lists aren't adequate for these purposes. Let's look at the advantages and shortcomings of several requirements-labeling methods. Select whichever technique makes the most sense for your situation.

Number 8, with a bullet

I was chatting with my seatmate on a long airplane flight once. It turned out that Dave was also in the software business. I mentioned that I had some interest in requirements. Dave pulled an SRS out of his briefcase. I don't know if he carried one with him everywhere he went for emergency purposes or what. I saw that the requirements in his document were organized hierarchically, but they were all in bulleted list form. He had up to eight levels of bullet hierarchy in some places. They all used different symbols—❍, ■, ◆, ✓, ❑, ➪, and the like—but they had no labels more meaningful than those simple symbols. It's impossible to refer to a bulleted item or to trace it to a design element, code segment, or test.

Sequence number

The simplest approach gives every requirement a unique sequence number, such as UC-9 or FR-26. Commercial requirements management tools assign such an identifier when a user adds a new requirement to the tool's database. The prefix indicates the requirement type, such as *FR* for *functional requirement*. A number is not reused if a requirement is deleted, so you don't have to worry about a reader confusing the original FR-26 with a new FR-26. This simple numbering approach doesn't provide any logical or hierarchical grouping of related requirements, the number doesn't imply any kind of ordering, and the labels give no clue as to what each requirement is about. It does make it easy to retain a unique identifier if you move requirements around in a document.

Hierarchical numbering

In the most commonly used convention, if the functional requirements appear in section 3.2 of your SRS, they will all have labels that begin with 3.2. More digits indicate a more detailed, lower-level requirement, so you know that 3.2.4.3 is a child requirement of 3.2.4. This method is simple, compact, and familiar. Your word processor can probably assign the numbers automatically. Requirements management tools generally also support hierarchical numbering.

However, hierarchical numbering poses some problems. The labels can grow to many digits in even a medium-sized SRS. Numeric labels tell you nothing about the intent of a requirement. If you are using a word processor, typically this scheme does not generate persistent labels. If you insert a new requirement, the numbers of the following requirements in that section all will be incremented. Delete or move a requirement, and the numbers following it in that section will be decremented. Delete, insert, merge, or move whole sections, and a lot of labels change. These changes disrupt any references to those requirements elsewhere in the system.

Trap A BA once told me in all seriousness, "We don't let people insert requirements—it messes up the numbering." Don't let ineffective practices hamper your ability to work effectively and sensibly.

An improvement over hierarchical numbering is to number the major sections of the requirements hierarchically and then identify individual functional requirements in each section with a short text code followed by a sequence number. For example, the SRS might contain "Section 3.5—Editor Functions," and the requirements in that section could be labeled ED-1, ED-2, and so forth. This approach provides some hierarchy and organization while keeping the labels short, somewhat meaningful, and less positionally dependent. It doesn't totally solve the sequence number problem, though.

Hierarchical textual tags

Consultant Tom Gilb (1988) suggests a text-based hierarchical tagging scheme for labeling individual requirements. Consider this requirement: "The system shall ask the user to confirm any request to print more than 10 copies." This requirement might be tagged Print.ConfirmCopies. This indicates that it is part of the print function and relates to the number of copies to print. Hierarchical textual

tags are structured, meaningful, and unaffected by adding, deleting, or moving other requirements. The sample SRS in Appendix C illustrates this labeling technique, as do other examples throughout the book. This method also is suitable for labeling business rules if you're maintaining them manually, rather than in a dedicated business rules repository or tool.

Using hierarchical textual tags like this helps solve another problem. With any hierarchical organization you have parent-child relationships between requirements. If the parent is written as a functional requirement, the relationship between the children and the parent can be confusing. A good convention is to write the parent requirement to look like a title, a heading, or a feature name, rather than looking like a functional requirement in itself. The children requirements of that parent, in the aggregate, deliver the capability described in the parent. Following is an example that contains a heading and four functional requirements.

Product: Ordering products from the website

.Cart	The website shall use a shopping cart to contain products a Customer selects to purchase.
.Discount	The shopping cart shall provide one discount code field. Each discount code provides either a specific discount percentage or a fixed dollar discount amount on specific items in the cart.
.Error	If the Customer enters an invalid discount code, the website shall display an error message.
.Shipping	The shopping cart shall add a shipping charge if a Customer orders a physical product that must be mailed.

The full unique ID of each requirement is built by appending each line's label to the parent labels above it. The **Product** statement is written as a heading, not as a discrete requirement. The first functional requirement is tagged **Product.Cart**. The full ID for the third requirement is **Product.Discount.Error**. This hierarchical scheme avoids the maintenance problems with the hierarchical numbering, but the tags are longer and you do have to think of meaningful names for them, perhaps building from the name of the relevant feature. It can be challenging to maintain uniqueness, especially if you have multiple people working on the set of requirements. You can simplify the scheme by combining the hierarchical naming technique with a sequence number suffix for small sets of requirements: Product.Cart.01, Product.Cart.02, and so on. Many schemes can work.

Dealing with incompleteness

Sometimes you know that you lack a piece of information about a specific requirement. Use the notation *TBD* (to be determined) to flag these knowledge gaps. Plan to resolve all TBDs before implementing a set of requirements. Any uncertainties that remain increase the risk of a developer or a tester making errors and having to perform rework. When the developer encounters a TBD, he might make his best guess—which won't always be correct—instead of tracking down the requirement's originator to resolve it. If you must proceed with construction of the next product increment while TBDs remain, either defer implementing the unresolved requirements or design those portions of the product to be easily modifiable when the open issues are resolved. Record TBDs and other requirements questions in an issues list. As the number of open issues dwindles, the requirements are stabilizing. Chapter 27 further describes managing and resolving open issues.

Trap TBDs won't resolve themselves. Number the TBDs, record who is responsible for resolving each issue and by when, review their status at regular checkpoints, and track them to closure.

User interfaces and the SRS

Incorporating user interface designs in the SRS has both benefits and drawbacks. On the plus side, exploring possible user interfaces with paper prototypes, working mock-ups, wireframes, or simulation tools makes the requirements tangible to both users and developers. As discussed in Chapter 15, "Risk reduction through prototyping," these are powerful techniques for eliciting and validating requirements. If the product's users have expectations of how portions of the product might look and feel—and hence could be disappointed if their expectations weren't fulfilled—those expectations belong in the realm of requirements.

On the negative side, screen images and user interface architectures describe solutions and might not truly be requirements. Including them in the SRS makes the document larger, and big requirements documents frighten some people. Delaying baselining of the SRS until the UI design is complete can slow down development and try the patience of people who are already concerned about spending too much time on requirements. Including UI design in the requirements can result in the visual design driving the requirements, which often leads to functional gaps. The people who write the requirements aren't necessarily well qualified for designing user interfaces. Additionally, after stakeholders see a user interface in an SRS (or anywhere else), they will not "unsee" it. Early visualization can clarify requirements, but it can also lead to resistance to improving the UI over time.

Screen layouts don't replace written user and functional requirements. Don't expect developers to deduce the underlying functionality and data relationships from screen shots. One Internet development company repeatedly got in trouble because the team routinely went directly from signing a contract with a client into an eight-hour visual design workshop. They never sufficiently understood what a user would be able to do at each website they built, so they spent a lot of time fixing the sites after delivery.

If you really do want to implement certain functionality with specific UI controls and screen layouts, it's both appropriate and important to include that information in the SRS as design constraints. Design constraints restrict the choices available to the user interface designer. Just make sure that you don't impose constraints unnecessarily, prematurely, or for the wrong reasons. If the SRS is specifying an enhancement to an existing system, it often makes sense to include screen displays exactly as they are to be implemented. The developers are already constrained by the current reality of the existing system, so it's possible to know up front just how the modified—and perhaps also the new—displays should look.

A sensible balance is to include conceptual images—I call them sketches, no matter how nicely drawn they are—of selected displays in the requirements without demanding that the implementation precisely follow those models. See Figure 10-1 for a sample webpage sketch. Incorporating such sketches in the SRS helpfully communicates another view of the requirements,

but makes it clear that the sketches are not the committed screen designs. For example, a preliminary sketch of a complex dialog box will illustrate the intent behind a group of requirements, but a visual designer might turn it into a tabbed dialog box to improve usability.

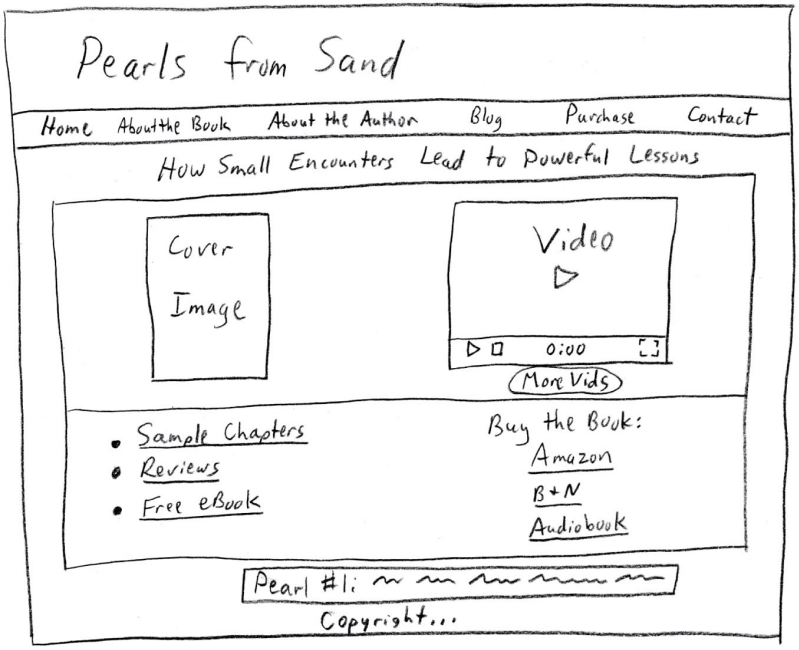

FIGURE 10-1 Example of a user interface "sketch" suitable for inclusion in a requirements document.

Teams working on projects that have many screens might find it more manageable to document the user interface design specifics in a separate user interface specification or by using UI design tools or prototyping tools. Use techniques such as display-action-response models to describe screen element names, their properties, and their behavior in detail (Beatty and Chen 2012).

A software requirements specification template

Every software development organization should adopt one or more standard SRS templates for its projects. Various SRS templates are available (for example: ISO/IEC/IEEE 2011; Robertson and Robertson 2013). If your organization tackles various kinds or sizes of projects, such as new, large system development as well as minor enhancements to existing systems, adopt an SRS template for each major project class. See the "Template tactics" sidebar in Chapter 5, "Establishing the business requirements," for some thoughts about how to use document templates effectively.

Figure 10-2 illustrates an SRS template that works well for many types of projects. Appendix C contains a sample SRS that follows this template. This template, with usage guidance embedded in each section, is available for downloading from this book's companion content website. Some people format such guidance text as "hidden text" in Microsoft Word. That way, you can leave the prompts in the document. If you want a memory jogger, just turn on nonprinting characters to see the information.

1. Introduction
- 1.1 Purpose
- 1.2 Document conventions
- 1.3 Project scope
- 1.4 References

2. Overall description
- 2.1 Product perspective
- 2.2 User classes and characteristics
- 2.3 Operating environment
- 2.4 Design and implementation constraints
- 2.5 Assumptions and dependencies

3. System features
- 3.x System feature X
  - 3.x.1 Description
  - 3.x.2 Functional requirements

4. Data requirements
- 4.1 Logical data model
- 4.2 Data dictionary
- 4.3 Reports
- 4.4 Data acquisition, integrity, retention, and disposal

5. External interface requirements
- 5.1 User interfaces
- 5.2 Software interfaces
- 5.3 Hardware interfaces
- 5.4 Communications interfaces

6. Quality attributes
- 6.1 Usability
- 6.2 Performance
- 6.3 Security
- 6.4 Safety
- 6.x [others]

7. Internationalization and localization requirements

8. Other requirements

Appendix A: Glossary

Appendix B: Analysis models

FIGURE 10-2 Proposed template for a software requirements specification.

Sometimes a piece of information could logically be recorded in several template sections. Pick one section and use it consistently for that kind of information on your project. Avoid duplicating information in multiple sections even if it could logically fit in more than one (Wiegers 2006). Cross-references and hyperlinks can help readers find the information they need.

When you create requirements documents, use effective version control practices and tools to make sure all readers know which version they are reading. Include a revision history to provide a record of changes made in the document, who made each change, when it was made, and the reason for it (see Chapter 27). The rest of this section describes the information to include in each section of the SRS.

Important You can incorporate material by reference to other existing project documents instead of duplicating information in the SRS. Hyperlinks between documents are one way to do this, as are traceability links defined in a requirements management tool. A risk with hyperlinks is that they can break if the document folder hierarchy changes. Chapter 18, "Requirements reuse," discusses several techniques for reusing existing requirements knowledge.

1. Introduction

The introduction presents an overview to help the reader understand how the SRS is organized and how to use it.

1.1 Purpose

Identify the product or application whose requirements are specified in this document, including the revision or release number. If this SRS pertains to only part of a complex system, identify that portion or subsystem. Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers.

1.2 Document conventions

Describe any standards or typographical conventions used, including the meaning of specific text styles, highlighting, or notations. If you are manually labeling requirements, you might specify the format here for anyone who needs to add one later.

1.3 Project scope

Provide a short description of the software being specified and its purpose. Relate the software to user or corporate goals and to business objectives and strategies. If a separate vision and scope or similar document is available, refer to it rather than duplicating its contents here. An SRS that specifies an incremental release of an evolving product should contain its own scope statement as a subset of the long-term strategic product vision. You might provide a high-level summary of the major features the release contains or the significant functions that it performs.

1.4 References

List any documents or other resources to which this SRS refers. Include hyperlinks to them if they are in a persistent location. These might include user interface style guides, contracts, standards, system requirements specifications, interface specifications, or the SRS for a related product. Provide enough information so that the reader can access each reference, including its title, author, version number, date, source, storage location, or URL.

2. Overall description

This section presents a high-level overview of the product and the environment in which it will be used, the anticipated users, and known constraints, assumptions, and dependencies.

2.1 Product perspective

Describe the product's context and origin. Is it the next member of a growing product line, the next version of a mature system, a replacement for an existing application, or an entirely new product? If this SRS defines a component of a larger system, state how this software relates to the overall system and identify major interfaces between the two. Consider including visual models such as a context diagram or ecosystem map (described in Chapter 5) to show the product's relationship to other systems.

2.2 User classes and characteristics

Identify the various user classes that you anticipate will use this product, and describe their pertinent characteristics. (See Chapter 6, "Finding the voice of the user.") Some requirements might pertain only to certain user classes. Identify the favored user classes. User classes represent a subset of the stakeholders described in the vision and scope document. User class descriptions are a reusable resource. If a master user class catalog is available, you can incorporate user class descriptions by simply pointing to them in the catalog instead of duplicating information here.

2.3 Operating environment

Describe the environment in which the software will operate, including the hardware platform; operating systems and versions; geographical locations of users, servers, and databases; and organizations that host the related databases, servers, and websites. List any other software components or applications with which the system must peacefully coexist. If extensive technical infrastructure work needs to be performed in conjunction with developing the new system, consider creating a separate infrastructure requirements specification to detail that work.

2.4 Design and implementation constraints

There are times when a certain programming language must be used, a particular code library that has already had time invested to develop it needs to be used, and so forth. Describe any factors that will restrict the options available to the developers and the rationale for each constraint. Requirements that incorporate or are written in the form of solution ideas rather than needs are imposing design constraints, often unnecessarily, so watch out for those. Constraints are described further in Chapter 14, "Beyond functionality."

2.5 Assumptions and dependencies

An *assumption* is a statement that is believed to be true in the absence of proof or definitive knowledge. Problems can arise if assumptions are incorrect, are obsolete, are not shared, or change, so certain assumptions will translate into project risks. One SRS reader might assume that the product will conform to a particular user interface convention, whereas another might assume something different. A developer might assume that a certain set of functions will be custom-written for this application, whereas the business analyst might assume that they will be reused from a previous project, and the project manager might expect to procure a commercial function library. The assumptions to include here are those related to system functionality; business-related assumptions appear in the vision and scope document, as described in Chapter 5.

Identify any *dependencies* the project or system being built has on external factors or components outside its control. For instance, if Microsoft .NET Framework 4.5 or a more recent version must be installed before your product can run, that's a dependency.

3. System features

The template in Figure 10-2 shows functional requirements organized by system feature, which is just one possible way to arrange them. Other organizational options include arranging functional requirements by functional area, process flow, use case, mode of operation, user class, stimulus, and response. Hierarchical combinations of these elements are also possible, such as use cases within user classes. There is no single right choice; select a method of organization that makes it easy for readers to understand the product's intended capabilities. We'll describe the feature scheme as an example.

3.x System feature X

State the name of the feature in just a few words, such as "3.1 Spell Check." Repeat section 3.x with its subsections 3.x.1 and 3.x.2 for each system feature.

3.x.1 Description

Provide a short description of the feature and indicate whether it is of high, medium, or low priority. (See Chapter 16, "First things first: Setting requirement priorities.") Priorities often are dynamic, changing over the course of the project. If you're using a requirements management tool, define a requirement attribute for priority. Requirement attributes are discussed in Chapter 27 and requirements management tools in Chapter 30.

3.x.2 Functional requirements

Itemize the specific functional requirements associated with this feature. These are the software capabilities that must be implemented for the user to carry out the feature's services or to perform a use case. Describe how the product should respond to anticipated error conditions and to invalid inputs and actions. Uniquely label each functional requirement, as described earlier in this chapter. If you're using a requirements management tool, you can create multiple attributes for each functional requirement, such as rationale, origin, and status.

4. Data requirements

Information systems provide value by manipulating data. Use this section of the template to describe various aspects of the data that the system will consume as inputs, process in some fashion, or create as outputs. Chapter 13, "Specifying data requirements," addresses this topic in more detail. Stephen Withall (2007) describes many patterns for documenting data (also known as information) requirements precisely.

4.1 Logical data model

As described in Chapter 13, a data model is a visual representation of the data objects and collections the system will process and the relationships between them. Numerous notations exist for data modeling, including entity-relationship diagrams and UML class diagrams. You might include a data model for the business operations being addressed by the system, or a logical representation for the data that the system will manipulate. This is not the same thing as an implementation data model that will be realized in the form of database design.

4.2 Data dictionary

The data dictionary defines the composition of data structures and the meaning, data type, length, format, and allowed values for the data elements that make up those structures. Commercial data modeling tools often include a data dictionary component. In many cases, you're better off storing the data dictionary as a separate artifact, rather than embedding it in the middle of an SRS. That also increases its reusability potential in other projects. Chapter 13 discusses the data dictionary.

4.3 Reports

If your application will generate any reports, identify them here and describe their characteristics. If a report must conform to a specific predefined layout, you can specify that here as a constraint, perhaps with an example. Otherwise, focus on the logical descriptions of the report content, sort sequence, totaling levels, and so forth, deferring the detailed report layout to the design stage. Chapter 13 offers guidance on specifying reports.

4.4 Data acquisition, integrity, retention, and disposal

If relevant, describe how data is acquired and maintained. For instance, when starting a data inventory feed, you might need to do an initial dump of all the inventory data to the receiving system and then have subsequent feeds that consist only of changes. State any requirements regarding the need to protect the integrity of the system's data. Identify any specific techniques that are necessary, such as backups, checkpointing, mirroring, or data accuracy verification. State policies the system must enforce for either retaining or disposing of data, including temporary data, metadata, residual data (such as deleted records), cached data, local copies, archives, and interim backups.

5. External interface requirements

This section provides information to ensure that the system will communicate properly with users and with external hardware or software elements. Reaching agreement on external and internal system interfaces has been identified as a software industry best practice (Brown 1996). A complex system with multiple subcomponents should create a separate interface specification or system architecture specification. The interface documentation could incorporate material from other documents by reference. For instance, it could point to a hardware device manual that lists the error codes that the device could send to the software.

Interface wars

Two software teams collaborated to build the A. Datum Corporation's flagship product. The knowledge base team built a complex inference engine in C++, and the applications team implemented the user interface in Java. The two subsystems communicated through an application programming interface (API). Unfortunately, the knowledge base team periodically modified the API, with the consequence that the complete system would not build and execute correctly. The applications team needed several hours to diagnose each problem they discovered and determine the root cause as being an API change. These changes were not agreed upon by the two teams, were not communicated to all affected parties, and were not coordinated with corresponding modifications in the Java code. A change in an interface *demands* communication with the person, group, or system on the other side of that interface. The interfaces glue your system components—including the users—together, so document the interface details and synchronize necessary modifications through your project's change control process.

5.1 User interfaces

Describe the logical characteristics of each user interface that the system needs. Some specific characteristics of user interfaces could appear in section 6.1 Usability. Some possible items to address here are:

- References to user interface standards or product line style guides that are to be followed
- Standards for fonts, icons, button labels, images, color schemes, field tabbing sequences, commonly used controls, branding graphics, copyright and privacy notices, and the like
- Screen size, layout, or resolution constraints
- Standard buttons, functions, or navigation links that will appear on every screen, such as a help button
- Shortcut keys
- Message display and phrasing conventions
- Data validation guidelines (such as input value restrictions and when to validate field contents)

- Layout standards to facilitate software localization
- Accommodations for users who are visually impaired, color blind, or have other limitations

5.2 Software interfaces

Describe the connections between this product and other software components (identified by name and version), including other applications, databases, operating systems, tools, libraries, websites, and integrated commercial components. State the purpose, formats, and contents of the messages, data, and control values exchanged between the software components. Specify the mappings of input and output data between the systems and any translations that need to be made for the data to get from one system to the other. Describe the services needed by or from external software components and the nature of the inter-component communications. Identify data that will be exchanged between or shared across software components. Specify nonfunctional requirements affecting the interface, such as service levels for response times and frequencies, or security controls and restrictions. Some of this information might be specified as data requirements in section 4 or as interoperability requirements in section 6, Quality attributes.

5.3 Hardware interfaces

Describe the characteristics of each interface between the software components and hardware components, if any, of the system. This description might include the supported device types, the data and control interactions between the software and the hardware, and the communication protocols to be used. List the inputs and outputs, their formats, their valid values or ranges, and any timing issues developers need to be aware of. If this information is extensive, consider creating a separate interface specification document. For more about specifying requirements for systems containing hardware, see Chapter 26, "Embedded and other real-time systems projects."

5.4 Communications interfaces

State the requirements for any communication functions the product will use, including email, web browser, network protocols, and electronic forms. Define any pertinent message formatting. Specify communication security and encryption issues, data transfer rates, handshaking, and synchronization mechanisms. State any constraints around these interfaces, such as whether certain types of email attachments are acceptable or not.

6. Quality attributes

This section specifies nonfunctional requirements other than constraints, which are recorded in section 2.4, and external interface requirements, which appear in section 5. These quality requirements should be specific, quantitative, and verifiable. Indicate the relative priorities of various attributes, such as ease of use over ease of learning, or security over performance. A rich specification notation such as Planguage clarifies the needed levels of each quality much better than can simple descriptive statements (see the "Specifying quality requirements with Planguage" section in Chapter 14). Chapter 14 presents more information about these quality attribute requirements and many examples.

6.1 Usability

Usability requirements deal with ease of learning, ease of use, error avoidance and recovery, efficiency of interactions, and accessibility. The usability requirements specified here will help the user interface designer create the optimum user experience.

6.2 Performance

State specific performance requirements for various system operations. If different functional requirements or features have different performance requirements, it's appropriate to specify those performance goals right with the corresponding functional requirements, rather than collecting them in this section.

6.3 Security

Specify any requirements regarding security or privacy issues that restrict access to or use of the product. These could refer to physical, data, or software security. Security requirements often originate in business rules, so identify any security or privacy policies or regulations to which the product must conform. If these are documented in a business rules repository, just refer to them.

6.4 Safety

Specify requirements that are concerned with possible loss, damage, or harm that could result from use of the product. Define any safeguards or actions that must be taken, as well as potentially dangerous actions that must be prevented. Identify any safety certifications, policies, or regulations to which the product must conform.

6.x [Others]

Create a separate section in the SRS for each additional product quality attribute to describe characteristics that will be important either to customers or to developers and maintainers. Possibilities include availability, efficiency, installability, integrity, interoperability, modifiability, portability, reliability, reusability, robustness, scalability, and verifiability. Chapter 14 describes a procedure for focusing on those attributes that are of most importance to a particular project.

7. Internationalization and localization requirements

Internationalization and localization requirements ensure that the product will be suitable for use in nations, cultures, and geographic locations other than those in which it was created. Such requirements might address differences in currency; formatting of dates, numbers, addresses, and telephone numbers; language, including national spelling conventions within the same language (such as American versus British English), symbols used, and character sets; given name and family name order; time zones; international regulations and laws; cultural and political issues; paper sizes used; weights and measures; electrical voltages and plug shapes; and many others. Internationalization and localization requirements could well be reusable across projects.

8. [Other requirements]

Define any other requirements that are not covered elsewhere in the SRS. Examples are legal, regulatory, or financial compliance and standards requirements; requirements for product installation, configuration, startup, and shutdown; and logging, monitoring, and audit trail requirements. Instead of just combining these all under "Other," add any new sections to the template that are pertinent to your project. Omit this section if all your requirements are accommodated in other sections. Transition requirements that are necessary for migrating from a previous system to a new one could be included here if they involve software being written (as for data conversion programs), or in the project management plan if they do not (as for training development or delivery).

Appendix A: Glossary

Define any specialized terms that a reader needs to know to understand the SRS, including acronyms and abbreviations. Spell out each acronym and provide its definition. Consider building a reusable enterprise-level glossary that spans multiple projects and incorporating by reference any terms that pertain to this project. Each SRS would then define only those terms specific to an individual project that do not appear in the enterprise-level glossary. Note that data definitions belong in the data dictionary, not the glossary.

Appendix B: Analysis models

This optional section includes or points to pertinent analysis models such as data flow diagrams, feature trees, state-transition diagrams, or entity-relationship diagrams. (See Chapter 12, "A picture is worth 1024 words.") Often it's more helpful for the reader if you incorporate certain models into the relevant sections of the specification instead of collecting them at the end.

Requirements specification on agile projects

Projects following agile development life cycles take a variety of approaches to specifying requirements that differ from the method just described. As you saw in Chapter 8, "Understanding user requirements," many agile projects employ user stories during elicitation. Each user story is a statement of a user need or functionality that will be valuable to the user or purchaser of the system (Cohn 2004; Cohn 2010). Teams might begin specification on agile projects by writing just enough information for each user story so that the stakeholders have a general understanding of what the story is about and can prioritize it relative to other stories. This allows the team to begin planning allocations of specific stories to iterations. The team might aggregate a group of related stories into a "minimally marketable feature" that needs to be fully implemented prior to a product release so the feature delivers the expected customer value.

User stories are accumulated and prioritized into a dynamic *product backlog* that evolves throughout the project. Large stories that encompass significant functionality that cannot be implemented within a single iteration are subdivided into smaller stories, which are allocated to multiple iterations for implementation. (See Chapter 20, "Agile projects.") User stories can be recorded on something as simple

as index cards, instead of in a traditional document. Some agile teams record their stories in a story management tool, whereas others don't retain them at all following implementation.

As the team gets into each iteration, conversations among the product owner, people performing the business analyst role, developers, testers, and users will flesh out the details of each story allocated to the iteration. That is, specification involves the progressive refinement of detail at the right stage of the project, which is a good practice on any project. Those details generally correspond to what we have identified as functional requirements in the SRS. However, agile projects often represent those details in the form of user acceptance tests that describe how the system will behave if the story is properly implemented. The tests for a story are conducted during the iteration in which the story is implemented and in future iterations for regression testing. As with all tests, they should cover exception conditions as well as the expected behavior. These acceptance tests can be written on cards as well or recorded in a more persistent form, such as in a testing tool. Tests should be automated to assure rapid and complete regression testing. If the team elects to discard the original user stories, then the only persistent documentation of the requirements is likely to be the acceptance tests, if they are stored in a tool.

Similarly, nonfunctional requirements can be written on cards not as user stories but as constraints (Cohn 2004). Alternatively, teams might specify nonfunctional requirements that are associated with a specific user story in the form of acceptance criteria or tests, such as to demonstrate achievement of specific quality attribute goals. As an example, security tests might demonstrate that certain users are permitted to access the functionality described in a particular user story but that the system blocks access for other users. The agile team is not precluded from using other methods to represent requirements knowledge, such as analysis models or a data dictionary. They should select whatever representation techniques are customary and appropriate for their culture and project.

It's up to each project team to choose the most appropriate forms for specifying its software requirements. Remember the overarching goal of requirements development: to accumulate a shared understanding of requirements that is *good enough* to allow construction of the next portion of the product to proceed at an acceptable level of risk. The appropriate level of formality and detail in which to document requirements depends on factors including the following:

- The extent to which just-in-time informal verbal and visual communication between customers and developers can supply the necessary details to permit the correct implementation of each user requirement
- The extent to which informal communication methods can keep the team effectively synchronized across time and space
- The extent to which it is valuable or necessary to retain requirements knowledge for future enhancement, maintenance, application reengineering, verification, statutory and audit mandates, product certification, or contractual satisfaction
- The extent to which acceptance tests can serve as effective replacements for descriptions of the expected system capabilities and behaviors
- The extent to which human memories can replace written representations

No matter what type of product the team is building, what development life cycle they are following, or what elicitation techniques the BA is using, effective requirements specification is an essential key to success. There are many ways to achieve this. Just remember that when you don't specify high-quality requirements, the resulting software is like a box of chocolates: you never know what you're going to get.

Next steps

- Review your project's set of requirements against the template in Figure 10-2 to see if you have requirements from all the sections that pertain to your project. This chapter is less about populating a specific template and more about ensuring that you accumulate the necessary information for a successful project; the template is a helpful reminder.
- If your organization doesn't already have a standard SRS template, convene a small working group to adopt one. Begin with the template in Figure 10-2 and adapt it to best meet the needs of your organization's projects and products. Agree on a convention for labeling individual requirements.
- If you are storing your requirements in some form other than in a traditional document, such as in a requirements management tool, study the SRS template in Figure 10-2 and see if there are any categories of requirements information that you are not currently eliciting and recording. Modify your repository to incorporate those categories so the repository can serve as a reminder for future requirements elicitation activities.