Chapter 2

Cosmic Truths About Software Requirements

In this chapter:

As every consultant knows, the correct answer to nearly any question regarding software is, "It depends." This isn't just a consultant's cop-out—it's true. The best advice for how to proceed in a given situation depends on the nature of the project, its constraints, the culture of the organization and team, the business environment, and other factors. But having worked with many organizations, I've made some observations about software requirements that really do seem to be universally applicable. This chapter presents some of these "cosmic truths" and their implications for the practicing requirements analyst.

Requirements Realities

Cosmic Truth #1: If you don't get the requirements right, it doesn't matter how well you execute the rest of the project.

Requirements are the foundation for all the project work that follows. I don't mean the initial SRS you come up with early in the project, but rather the full set of requirements knowledge that is developed incrementally during the course of the project.

The purpose of a software development project is to build a product that provides value to a particular set of customers. Requirements development attempts to determine the mix of product capabilities and characteristics that will best deliver this customer value. This understanding evolves over time as customers provide feedback on the early work and refine their expectations and needs. If this set of expectations isn't adequately explored and crafted into a set of product features and attributes, the chance of satisfying customer needs is slim.

As mentioned in the previous chapter, requirements validation is one of the vital subcomponents of requirements development, along with elicitation, analysis, and specification. Validation involves demonstrating that the specified requirements will meet customer needs. One useful technique for validating requirements is to work with suitable customer representatives to develop *user acceptance criteria*. These criteria define how customers determine whether they're willing to pay for the product or to begin using it to do their work. User acceptance criteria typically stipulate that the product allows the users to properly perform their most significant tasks, handles the common error conditions, and satisfies the users' quality expectations. User acceptance criteria aren't a substitute for thorough system testing. They do, however, provide a necessary perspective to determine whether the requirements are indeed right.

Cosmic Truth #2: Requirements development is a discovery and invention process, not just a collection process.

People often talk about "gathering requirements." This phrase suggests that the requirements are just lying around waiting to be picked like flowers or to be sucked out of the users' brains by the analyst. I prefer the term *requirements elicitation* to *requirements gathering*. Elicitation includes some discovery and some invention, as well as recording those bits of requirements information that customer representatives and subject matter experts offer to the analyst. Elicitation demands iteration. The participants in an elicitation discussion won't think of everything they'll need up front, and their thinking will change as the project continues. Requirements development is an exploratory activity.

The analyst is not simply a scribe who records what customers say. The analyst is an investigator who asks questions that stimulate the customers' thinking, seeking to uncover hidden information and generate new ideas. (See Chapter 7, "An Inquiry, Not an Inquisition.") It's fine for an analyst to propose requirements that might meet customer needs, provided that customers agree that those requirements add value before they go into the product (Robertson 2002). An analyst might ask a customer, "Would it be helpful if the system could do <whatever idea he has>?" The customer might reply, "No, that wouldn't do much for us." Or the customer might reply, "You could do that? Wow, that would be great! We didn't even think to ask for that feature, but if you could build it in, it would save our users a lot of time." This creativity is part of the value that the analyst adds to the requirements conversation. Just be careful that analysts and developers don't attempt to define a product from the bottom up through suggested product features, rather than basing the requirements on an understanding of stakeholder goals and a broad definition of success.

Cosmic Truth #3: Change happens.

It's inevitable that requirements will change. Business needs evolve, new users or markets are identified, business rules and government regulations are revised, and operating environments change over time. In addition, the business need becomes clearer as the key stakeholders become better educated about what their true needs are.

The objective of a change control process is not to inhibit change. Rather, the objective is to *manage* change to ensure that the project incorporates the right changes for the right reasons. You need to anticipate and accommodate changes to produce the minimum disruption and cost to the project and its stakeholders. However, excessive churning of the requirements after they've been agreed upon suggests that elicitation was incomplete or ineffective—or that agreement was premature. (See Chapter 18, "The Line in the Sand.")

To help make change happen, establish a change control process. You can download a sample from my Web site, *http://www.processimpact.com/goodies.shtml*. When I helped to implement a change control process in an Internet development group at Eastman Kodak Company, the team members properly viewed it as a structure, not as a barrier (Wiegers 1999). The group found this process invaluable for dealing with its mammoth backlog of change requests.

Every project team also needs to determine who will be evaluating requested changes and making decisions to approve or reject them. This group is typically called the change (or configuration) control board, or CCB. A CCB should write a charter that defines its composition, scope of authority, operating procedures, and decision-making process. A template for such a charter is available from *http://www.processimpact.com/goodies.shtml*.

Nearly every software project becomes larger than originally anticipated, so expect your requirements to grow over time. According to consultant Capers Jones (2000), requirements growth typically averages 1 to 3 percent per month during design and coding. This can have a significant impact on a long-term project. To accommodate some expected growth, build contingency buffers—also known as management reserve—into your project schedules (Wiegers 2002b). These buffers will keep your commitments from being thrown into disarray with the first change that comes along.

I once spoke with a manager on a five-year project regarding requirements growth. I pointed out that, at an average growth rate of 2 percent per month, his project was likely to be more than double the originally estimated size by the end of the planned 60-month schedule. The manager agreed that this was a possibility. When I asked if his plans anticipated this growth potential, he gave the answer I expected: No. I'm highly skeptical that this project will be completed without enormous cost and schedule overruns.

When you know that requirements are uncertain and likely to change, use an incremental or iterative development life cycle. Don't attempt to get all the requirements "right" up front and freeze them. Instead, specify and *baseline* the first set of requirements based on what is known at the time. A baseline is a statement about the state of the requirements at a specific point in time, such as "We believe that these requirements will meet customer needs and are a suitable foundation for proceeding with design and construction." Then implement that fraction of the product, get some customer feedback, and move on to the next slice of functionality. This is the intent behind agile development methodologies, the spiral model, iterative prototyping, evolutionary delivery, and other incremental approaches to software development.

Finally, recognize that change always has a price. Even the act of reviewing a proposed change and rejecting it consumes time. Software people need to educate their project stakeholders so they understand that, sure, we can make that change you just requested, and here's what it's going to cost. Then the stakeholders can make appropriate business decisions about which desired changes should be incorporated and at what time.

Requirements Stakeholders

Cosmic Truth #4: The interests of all the project stakeholders intersect in the requirements process.

Consultant Tim Lister once defined project success as "meeting the set of all requirements and constraints held as expectations by key stakeholders." A *stakeholder* is an individual or group who is actively involved in the project, who is affected by the project, or who can influence its outcome. Figure 2-1 identifies some typical software project stakeholder groups. Certain stakeholders are internal to the project team, such as the project manager, developers, testers, and requirements analysts. Others are external, including customers who select, specify, or fund products; users who employ the systems; compliance certifiers; auditors; and marketing, manufacturing, sales, and support groups. The requirements analyst has a central communication role, being responsible for interacting with all these stakeholders. Further, the analyst is responsible for seeing that the system being defined will be fit for use by all stakeholders, perhaps working with a system architect to achieve this goal.

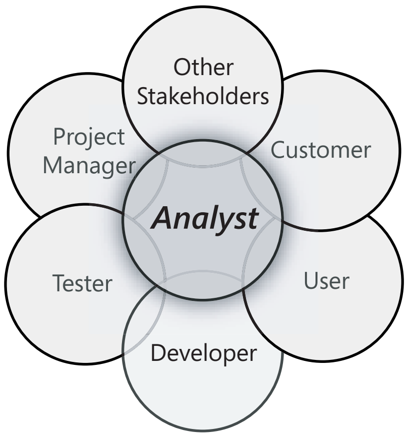

Figure 2-1 Some typical software project stakeholders.

At the beginning of your project, identify your key stakeholder groups and determine which individuals will represent the interests of each group. You can count on stakeholders having conflicting interests that must be reconciled. They can't all have veto power over each other. You need to identify early on the decision makers who will resolve these conflicts, and these decision makers must determine what their decision-making process will be. As my colleague Christian Fahlbusch, a seasoned project manager, points out, "I have found that there is usually one primary decision maker on a project, oftentimes the key sponsor within the organization. I don't rest until I have identified that person, and then I make sure he is always aware of the project's progress."

Cosmic Truth #5: Customer involvement is the most critical contributor to software quality.

Various studies have confirmed that inadequate customer involvement is a leading cause of the failure of software projects. Customers often claim they can't spend time working on requirements. However, customers who aren't happy because the delivered product missed the mark always find plenty of time to point out the problems. The development team is going to get the customer input it needs eventually. It's a lot cheaper—and a lot less painful—to get that input early on, rather than after the project is ostensibly done.

Customer involvement requires more than a workshop or two early in the project. Ongoing engagement by suitably empowered and enthusiastic customer representatives is a critical success factor for software development. Following are some good practices for engaging customers in requirements development:

- **Identify user classes.** Customers are a subset of stakeholders, and users are a subset of customers. You can further subdivide your user community into multiple *user classes* that have largely distinct needs (Gause and Lawrence 1999). Unrepresented user classes are likely to be disappointed with the project outcome.
- **Select product champions.** You need to determine who will be the literal voice of the customer for each user class. I call these people *product champions*. Ideally, product champions are actual users who represent their user-class peers. See Chapter 6, "The Myth of the On-Site Customer," for more about product champions.
- **Build prototypes.** Prototypes provide opportunities for user representatives to interact with a simulation or portion of the ultimate system. (See Chapter 13 of *Software Requirements, Second Edition*.) Prototypes are far more tangible than written requirements specifications. However, prototypes aren't a substitute for documenting the detailed requirements.
- **Agree on customer rights and responsibilities.** People who must work together rarely discuss the nature of their collaboration. The analyst should negotiate with the customer representatives early in the project to agree on the responsibilities each party has with respect to the requirements process. An agreed-upon collaboration strategy is a strong contributor to the participants' mutual success. See Chapter 2 of *Software Requirements, Second Edition* for some suggestions of customer rights and responsibilities in the requirements process.

Cosmic Truth #6: The customer is not always right, but the customer always has a point.

It's popular in some circles to do whatever any customer demands, claiming "The customer is always right." Of course, the customer is *not* always right! Sometimes customers are in a bad mood, uninformed, or unreasonable. If you receive conflicting input from multiple customers, which one of those customers is "always right"?

The customer may not always be right, but the analyst needs to understand and respect whatever point each customer is trying to make through his request for certain product features or attributes. The analyst needs to be alert for situations in which the customer could be in the wrong. Rather than simply promising anything a customer requests, strive to understand the rationale behind the customer's thinking and negotiate an acceptable outcome. Following are some examples of situations in which a customer might not be right:

- Presenting solutions in the guise of requirements.
- Failing to prioritize requirements or expecting the loudest voice to get top priority.
- Not communicating business rules and other constraints, or trying to get around them.
- Expecting a new software system to drive business-process changes.
- Not supplying appropriate representative users to participate in requirements elicitation.
- Failing to make decisions when analysts or developers need issues resolved.
- Not accepting the need for tradeoffs in both functional and nonfunctional requirements.
- Demanding impossible commitments.
- Not accepting the cost of change.

Requirements Specifications

Cosmic Truth #7: The first question an analyst should ask about a proposed new requirement is, "Is this requirement in scope?"

Anyone who's been in the software business for long has worked on a project that has suffered from scope creep. It is normal and often beneficial for requirements to grow over the course of a project. Scope creep, though, refers to the uncontrolled and continuous increase in requirements that makes it impossible to deliver a product on schedule.

To control scope creep, you need to have the project stakeholders agree on a scope definition, a boundary between the desired capabilities that lie within the scope for a given product release and those that do not. (See Chapter 17, "Defining Project Scope," for some scope-definition techniques.) Then, whenever some stakeholder proposes a new functional requirement, feature, or use case, the analyst can ask, "Is this in scope?" To help answer this question, some project teams write their scope definition on a large piece of cardstock, laminate it, and bring it to their requirements elicitation discussions.

If a specific requirement is deemed out of scope one week, in scope the next, then out of scope again later, the project's scope boundary is not clearly defined. And that's an open invitation to scope creep.

Cosmic Truth #8: Even the best requirements document cannot—and should not—replace human dialogue.

Even the best requirements specification won't contain every bit of information the developers and testers need to do their jobs. There will always be tacit knowledge that the stakeholders assume (rightly or wrongly) that other participants already know, along with the explicit knowledge that must be documented in the SRS. Analysts and developers will always need to talk with knowledgeable users and subject matter experts to refine details, clarify ambiguities, and fill in the blanks. This is the rationale behind having some key customers, such as product champions, work intimately with the analysts and developers throughout the project. The person performing the role of requirements analyst (even if this is one of the developers) should coordinate these discussions to make sure that all the participants reach the same understanding so that the pieces all fit together properly. A written specification is still valuable and necessary, though. A documented record of what stakeholders agreed to at a point in time is more reliable than human memory.

You need more detail in the requirements specifications if you aren't going to have opportunities for frequent conversations with user representatives and other decision makers. (See Chapter 13, "How Much Detail Do You Need?") A good example of this is when you're outsourcing the implementation of a requirements specification that your team created. Expect to spend considerable time on review cycles to clarify and agree on what the requirements mean. Also expect delays in getting questions answered and decisions made, which can slow down the entire project. This very issue was a major contributing factor in a lawsuit I know of between a software package vendor and a customer (Wiegers 2003b). The vendor allowed no time in the schedule for review following some requirements elicitation workshops, planning instead to begin construction immediately. Months later, many key requirements issues had not yet been resolved and the actual project status didn't remotely resemble the project plan.

Cosmic Truth #9: The requirements might be vague, but the product will be specific.

Specifying requirements precisely is hard! You're inventing something new, and no one is exactly sure what the product should be and do. People sometimes are comfortable with vague requirements. Customers might like them because it means they can redefine those requirements later on to mean whatever they want them to mean at any given moment. Developers sometimes favor vague requirements because they allow the developers to build whatever they want to build. This is all great fun, but it doesn't lead to high-quality software.

Ultimately, you are building only one product, and someone needs to decide just what that product will be. If customers and analysts don't make the decisions, the developers will be forced to. This is a sign that the key stakeholders are abdicating their responsibility to make requirements-level decisions, leaving those decisions to people who know far less about the problem.

Don't use uncertainty as an excuse for lack of precision. Acknowledge the uncertainty and find ways to address it, such as through prototyping. A valuable adjunct to simply specifying each requirement is to define *fit criteria* that a user or tester could employ to judge whether the requirement was implemented correctly and as intended (Robertson and Robertson 1999). Attempting to write such fit criteria will quickly reveal whether a requirement is stated precisely enough to be verifiable.

Cosmic Truth #10: You're never going to have perfect requirements.

Requirements are never finished or complete. There is no way to know for certain that you haven't overlooked some requirement, and there will always be some requirements that the analyst won't feel it is necessary to record. Rather than declaring the requirements "done" at some point, define a baseline. (See Chapter 18.) Once you've established a baseline, follow your change control process to modify the requirements, recognizing the implications of making changes. It's folly to think you can freeze the requirements and allow no changes after some initial elicitation activities.

Striving for perfection can lead to analysis paralysis. Analysis paralysis, in turn, can have a backlash effect. Stakeholders who have been burned once by a project that got mired in requirements issues are reluctant to invest in requirements development on their next project.

You don't succeed in business by writing a perfect SRS. From a pragmatic perspective, requirements development strives for requirements that are *good enough* to allow the team to proceed with design, construction, and testing at an acceptable level of risk. The risk is the threat of having to do expensive and unnecessary rework. Have team members who will need to base their own work on the requirements review them to judge whether they provide a suitably solid foundation for that subsequent work. Keep this practical goal of "good enough" in mind as you pursue your quest for quality requirements.