CHAPTER 27

Requirements management practices

"I finally finished implementing the multivendor catalog query feature," Shari reported at the Chemical Tracking System's weekly project status meeting. "Man, that was a lot of work!"

"Oh, the customers canceled that feature two weeks ago," the project manager, Dave, replied. "Didn't you get the revised SRS?"

Shari was confused. "What do you mean, it was canceled? Those requirements are at the top of page 6 of my latest SRS."

Dave said, "Hmmm, they're not in my copy. I've got version 1.5 of the SRS. What version are you looking at?"

"Mine says version 1.5 also," said Shari in disgust. "These documents should be identical, but obviously they're not. So, is this feature still needed, or did I just waste 30 hours of my life?"

If you've ever heard a conversation like this one, you know how frustrating it is when people waste time working from obsolete or inconsistent requirements specifications. Having great requirements gets you only partway to a solution; they also have to be well managed and effectively communicated among the project participants. Version control of individual requirements and sets of requirements is one of the core activities of requirements management.

Chapter 1, "The essential software requirement," divided the domain of software requirements engineering into requirements development and requirements management. (Some people refer to the entire domain as "requirements management," but we favor a narrower definition of that term.) This chapter addresses some principles and practices of requirements management. The other chapters in Part IV describe certain requirements management practices in more detail, including change control (Chapter 28, "Change happens"), change impact analysis (also Chapter 28), and requirements tracing (Chapter 29, "Links in the requirements chain"). Part IV concludes with a discussion of commercial tools that can help a project team develop and manage its requirements (Chapter 30, "Tools for requirements engineering"). Note that a project might be managing certain sets of agreed-upon requirements while concurrently performing requirements development activities on other portions of the product's requirements.

Requirements management process

Requirements management includes all activities that maintain the integrity, accuracy, and currency of requirements agreements throughout the project. Figure 27-1 shows the core activities of requirements management in four major categories: version control, change control, requirements status tracking, and requirements tracing.

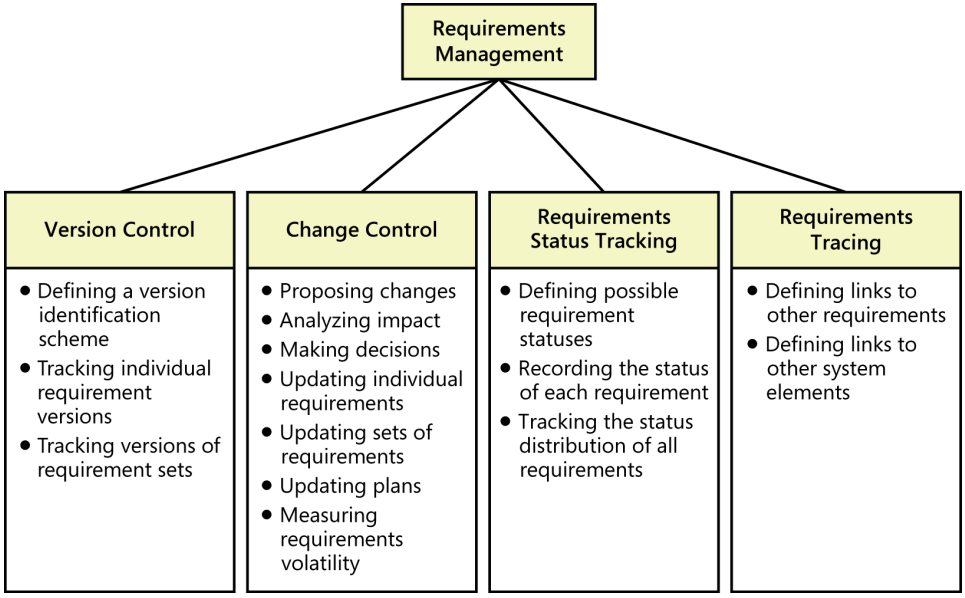

FIGURE 27-1 Major requirements management activities.

Your organization should define the activities that project teams are expected to perform to manage their requirements. Documenting these activities and training practitioners in their performance enables the members of the organization to conduct them consistently and effectively. Consider addressing the following topics:

- Tools, techniques, and conventions for distinguishing versions of individual requirements and of requirements sets
- The way that sets of requirements are approved and baselined (see Chapter 2, "Requirements from the customer's perspective")
- The ways that new requirements and changes to existing ones are proposed, evaluated, negotiated, and communicated
- How to assess the impact of a proposed change
- Requirement attributes and requirements status-tracking procedures, including the requirement statuses that you will use and who can change them
- Who is responsible for updating requirements trace information and when

- How to track and resolve requirements issues
- How the project's plans and commitments will reflect requirements changes
- How to use the requirements management (RM) tool effectively

You can include all this information in a single requirements management process description. Alternatively, you might prefer to write separate version control, change control, impact analysis, and status tracking procedures. These procedures should apply across your organization because they represent common functions that every project team ought to perform. Chapter 31, "Improving your requirements processes," describes several useful process assets for requirements management.

Your process descriptions should identify the team role that owns each of the requirements management activities. The project's business analyst typically has the lead responsibility for requirements management. The BA will set up the requirements storage mechanisms, define requirement attributes, coordinate requirement status and trace data updates, and monitor change activity as needed. The process description should also indicate who has authority to modify the requirements management process, how exceptions should be handled, and the escalation path for impediments encountered.

> **Trap** If no one on the project has responsibility for performing requirements management activities, don't expect them to get done. Similarly, if "everyone" has the responsibility, each person might expect that someone else is covering the necessary activities, so they can easily be overlooked.

The requirements baseline

Requirements development involves activities to elicit, analyze, specify, and validate a software project's requirements. Requirements development deliverables include business requirements, user requirements, functional and nonfunctional requirements, a data dictionary, and various analysis models. After they are reviewed and approved, any defined subset of these items constitutes a requirements baseline. As was described in Chapter 2, a requirements *baseline* is a set of requirements that stakeholders have agreed to, often defining the contents of a specific planned release or development iteration. The project might have additional agreements regarding deliverables, constraints, schedules, budgets, transition requirements, and contracts; those lie beyond the scope of this book.

At the time a set of requirements is baselined—typically following review and approval—the requirements are placed under configuration (or change) management. Subsequent changes can be made only through the project's defined change control procedure. Prior to baselining, the requirements are still evolving, so there's no point in imposing unnecessary process overhead on those modifications. A baseline could consist of some or all the requirements in a particular SRS (whether for an entire product or a single release), or a designated set of requirements stored in an RM tool, or an agreed-on set of user stories for a single iteration on an agile project.

If the scope of a release changes, update the requirements baseline accordingly. Distinguish the requirements in a particular baseline from others that were proposed but not accepted, are allocated to a different baseline, or remain unallocated in the product backlog. If the requirements are specified in the form of a document such as an SRS, clearly identify it as a baseline version to distinguish it from prior drafts. Storing requirements in an RM tool facilitates the identification of those that belong to a specific baseline and the management of changes to that baseline.

A development team that accepts proposed requirement changes or additions might not be able to fulfill its existing schedule and quality commitments. The project manager must negotiate changes to those commitments with affected managers, customers, and other stakeholders. The project can accommodate new or changed requirements in various ways:

- By deferring lower-priority requirements to later iterations or cutting them completely
- By obtaining additional staff or outsourcing some of the work
- By extending the delivery schedule or adding iterations to an agile project
- By sacrificing quality to ship by the original date

No single approach is universally correct, because projects differ in their flexibility of features, staff, budget, schedule, and quality (Wiegers 1996). The choice should be based on the project's business objectives and the priorities the key stakeholders established during project initiation. No matter how you respond to changing requirements, accept the reality of adjusting expectations and commitments when necessary. This is better than imagining that somehow all the new features will be incorporated by the original delivery date without budget overruns, team member burnout, or quality compromises.

Requirements version control

Version control—uniquely identifying different versions of an item—applies at the level of both individual requirements and requirements sets, most commonly represented in the form of documents. Begin version control as soon as you draft a requirement or a document so you can retain a history of changes made.

Every version of the requirements must be uniquely identified. Every team member must be able to access the current version of the requirements. Changes must be clearly documented and communicated to everyone affected. To minimize confusion and miscommunication, permit only designated individuals to update the requirements, and make sure that the version identifier changes whenever an update is made. Each circulated version of a requirements document or each requirement in a tool should include a revision history that identifies the changes made, the date of each change, the individual who made the change, and the reason for each change.

It's not a bug; it's a feature!

A contract development team received a flood of bug reports from the testers of the latest release they had just delivered to a customer. The contract team was perplexed—the system had passed all their own tests. After considerable investigation, it turned out that the customer was testing the new software against an obsolete version of the SRS. What the testers were reporting as bugs truly were features. Normally, this is just a little joke that software people like to make. The testers spent considerable time rewriting the tests against the correct version of the SRS and retesting the application, all because of a version control problem. Another colleague who once experienced the same kind of testing confusion because of an uncommunicated change said, "We probably wasted four to six hours of effort that our department had to absorb and couldn't spend on actual billable hours. I think software professionals would be shocked if they multiplied out these wasted hours times their bill rate to see what the loss in revenue is."

Similar confusion can arise when multiple BAs are working on a project. One BA begins to edit version 1.2 of the requirements specification. A few days later, another BA starts to work on some requirements and also labels his version 1.2, not knowing about the conflict. Pretty soon changes are lost, requirements are no longer up to date, work is overwritten, and confusion ensues.

The most robust approach to version control is to store the requirements in a requirements management tool, as described in Chapter 30. RM tools track the history of changes made to every requirement, which is valuable when you need to revert to an earlier version. Such a tool allows for comments describing the rationale behind a decision to add, modify, or delete a requirement. These comments are helpful if the requirement becomes a topic for discussion again in the future.

If you're storing requirements in documents, you can track changes by using the word processor's revision marks feature. This feature visually highlights changes made in the text with notations such as strikethrough highlighting for deletions and underscores for additions. When you baseline a document, first archive a marked-up version, then accept all the revisions, and then store the now clean version as the new baseline, ready for the next round of changes. Store requirements documents in a version control tool, such as the one your organization uses for controlling source code through check-out and check-in procedures. This will let you revert to earlier versions if necessary and to know who changed each document, when, and why. (Incidentally, this describes exactly how we wrote this book. We wrote the chapters in Microsoft Word, using revision marks as we iterated on the chapters. We had to refer back to previous versions on several occasions.)

I know of one project that stored several hundred use case documents written in Microsoft Word in a version control tool. The tool let the team members access all previous versions of every use case, and it logged the history of changes made to each one. The project's BA and her backup person had read-write access to the documents stored in the tool; the other team members had read-only access. This approach worked well for this team.

The simplest version control mechanism is to manually label each revision of a document according to a standard convention. Schemes that try to differentiate document versions based on dates are prone to confusion. I use a convention that labels the first version of any new document with its title and "Version 1.0 draft 1." The next draft keeps the same title but is identified as " Version 1.0 draft 2." The author increments the draft number with each iteration until the document is approved and baselined. At that time, the version identifier is changed to "Version 1.0 approved," again keeping the same document title. The next version is either "Version 1.1 draft 1" for a minor revision or "Version 2.0 draft 1" for a major change. (Of course, "major" and "minor" are subjective and depend on the context.) This scheme clearly distinguishes between draft and baselined document versions, but it does require manual discipline on the part of those who modify the documents.

Requirement attributes

Think of each requirement as an object with properties that distinguish it from other requirements. In addition to its textual description, each requirement should have supporting pieces of information or *attributes* associated with it. These attributes establish a context and background for each requirement. You can store attribute values in a document, a spreadsheet, a database, or—most effectively—a requirements management tool. It's cumbersome to use more than a couple of requirements attributes with documents.

RM tools typically provide several system-generated attributes in addition to letting you define others, some of which can be automatically populated. The tools let you query the database to view selected subsets of requirements based on their attribute values. For instance, you could list all high-priority requirements that were assigned to Shari for implementation in release 2.3 and have a status of Approved. Following is a list of potential requirement attributes to consider:

- Date the requirement was created
- Current version number of the requirement
- Author who wrote the requirement
- Priority
- Status
- Origin or source of the requirement
- Rationale behind the requirement
- Release number or iteration to which the requirement is allocated
- Stakeholder to contact with questions or to make decisions about proposed changes
- Validation method to be used or acceptance criteria

Wherefore this requirement?

The product manager at a company that makes electronic measurement devices wanted to track which requirements the team included simply because a competitor's product had the same capability. A good way to note such features is with a Rationale attribute, which indicates why a specific requirement is included in the product. Suppose you included some requirement because it meets the need of a particular user group. Later on, your marketing department decides they don't care about that user group any more. Having the justification present as a requirement attribute would help people decide whether that requirement could be omitted.

Another BA described his quandary with requirements that had no obvious justification. He said, "In my experience, many requirements exist without a real need behind them. They are introduced because the customer lacks an understanding of the technology, or because some key stakeholders get excited about the technology and want to show off, or because our sales team intentionally or unintentionally has misled the customer." If you can't provide a convincing rationale for a requirement and trace it back to a business need, the BA should question whether there's a real reason to devote effort to it.

Trap Selecting too many requirements attributes can overwhelm a team. They won't supply all attribute values for all requirements and won't use the attribute information effectively. Start with perhaps three or four key attributes. Add others only when you know how they will add value.

The requirements planned for a release will change as new requirements are added and existing ones are deleted or deferred. The team might be juggling separate requirements documents for multiple releases or iterations. Leaving obsolete requirements in the SRS can confuse readers as to whether those requirements are included in that baseline. A solution is to store the requirements in an RM tool and define a Release Number attribute. Deferring a requirement means changing its planned release, so simply updating the release number shifts the requirement into a different baseline. Handle deleted and rejected requirements by using a status attribute, as described in the next section.

Defining and updating these attribute values is part of the cost of requirements management, but that investment can yield a significant payback. One company periodically generated a requirements report that showed which of the 750 requirements from 3 related specifications were assigned to each designer. One designer discovered several requirements that she didn't realize were her responsibility. She estimated that she saved one to two months of engineering design rework that would have been required had she not found out about those requirements until later in the project. The larger the project, the easier it is to experience time-wasting miscommunications.

Tracking requirements status

"How are you coming on implementing that subsystem, Yvette?" asked the project manager, Dave.

"Pretty good, Dave. I'm about 90 percent done."

Dave was puzzled. "Didn't you say you were 90 percent done a couple of weeks ago?" he asked.

Yvette replied, "Yes, I thought I was, but now I'm really *90 percent done."*

Like nearly everyone, software developers are sometimes overly optimistic when they report how much of a task is complete. The common "90 percent done" syndrome doesn't tell Dave much about how close Yvette really is to finishing the subsystem. But suppose Yvette had replied, "Pretty good, Dave. Of the 84 requirements for the subsystem, 61 are implemented and verified, 14 are implemented but not yet verified, and I haven't implemented the other 9 yet." Tracking the status of each functional requirement throughout development provides a more precise gauge of project progress.

Status was one of the requirement attributes proposed in the previous section. Tracking status means comparing where you really are at a particular time against the expectation of what "complete" means for this development cycle. You might have planned to implement only certain flows of a use case in the current release, leaving full implementation for a future release. Monitor the status of just those functional requirements that were committed for the current release, because that's the set that's supposed to be 100 percent done before you declare success and ship the release.

> **Trap** There's an old joke that the first half of a software project consumes the first 90 percent of the resources and the second half consumes the other 90 percent of the resources. Overoptimistic estimation and overgenerous status tracking constitute a reliable formula for project overruns.

Table 27-1 lists several possible requirement statuses. Some practitioners add others, such as Designed (the design elements that address the functional requirement have been created and reviewed) and Delivered (the software containing the requirement is in the hands of the users, as for acceptance or beta testing). It's valuable to keep a record of rejected requirements and the reasons they were rejected. Rejected requirements have a way of resurfacing later during development or on a future project. The Rejected status lets you keep a proposed requirement available for possible future reference without cluttering up a specific release's set of committed requirements. You don't need to monitor all of the possible statuses in Table 27-1; choose the ones that add value to your requirements activities.

TABLE 27-1 Suggested requirement statuses

Status	Definition
Proposed	The requirement has been requested by an authorized source.
In Progress	A business analyst is actively working on crafting the requirement.
Drafted	The initial version of the requirement has been written.
Approved	The requirement has been analyzed, its impact on the project has been estimated, and it has been allocated to the baseline for a specific release. The key stakeholders have agreed to incorporate the requirement, and the software development group has committed to implement it.
Implemented	The code that implements the requirement has been designed, written, and unit tested. The requirement has been traced to the pertinent design and code elements. The software that implemented the requirement is now ready for testing, review, or other verification.
Verified	The requirement has satisfied its acceptance criteria, meaning that the correct functioning of the implemented requirement has been confirmed. The requirement has been traced to pertinent tests. It is now considered complete.
Deferred	An approved requirement is now planned for implementation in a later release.
Deleted	An approved requirement has been removed from the baseline. Include an explanation of why and by whom the decision was made to delete it.
Rejected	The requirement was proposed but was never approved and is not planned for implementation in any upcoming release. Include an explanation of why and by whom the decision was made to reject it.

Classifying requirements into several status categories is more meaningful than trying to monitor the percent completion of each requirement or of the complete release baseline. Update a requirement's status only when specified transition conditions are satisfied. Certain status changes also require updates to the requirements trace data to indicate which design, code, and test elements addressed the requirement, as illustrated in Table 29-1 in Chapter 29.

Figure 27-2 illustrates how you can visually monitor the status of a set of requirements throughout a hypothetical 10-month project. It shows the percentage of all the system's requirements having each status value at the end of each month. Tracking the distribution by percentages doesn't show whether the number of requirements in the baseline is changing over time. The number of requirements increases as scope is added and decreases when functionality is removed from the baseline. The curves illustrate how the project is approaching its goal of complete verification of all approved requirements. A body of work is done when all requirements allocated to it have a status of Verified, Deleted, or Deferred.

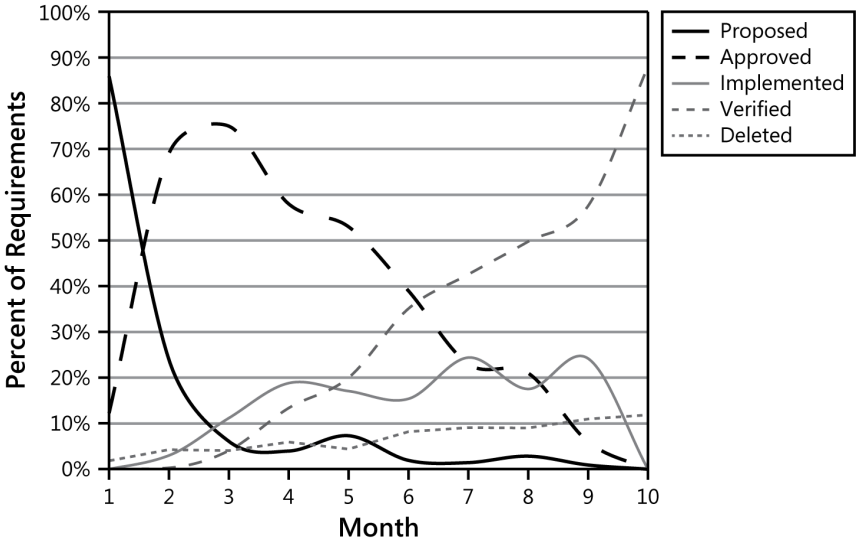

FIGURE 27-2 Tracking the distribution of requirements status throughout a project's development cycle.

Resolving requirements issues

Numerous questions, decisions, and issues related to requirements will arise during the course of a project. Potential issues include items flagged as TBD, pending decisions, information that is needed, and conflicts awaiting resolution. It's easy to lose sight of these open issues. Record issues in an issue-tracking tool so all affected stakeholders have access to them. Keep the issue-tracking and resolution process simple to ensure that nothing slips through the cracks. Some of the benefits of using an issue-tracking tool are:

- Issues from multiple requirements reviews are collected so that no issue ever gets lost.
- The project manager can easily see the current status of all issues.
- A single owner can be assigned to each issue.
- The history of discussion around an issue can be retained.
- The team can begin development earlier with a known set of open issues rather than having to wait until the SRS is complete.

Resolve requirements issues so they don't impede the timely baselining of a high-quality requirements set for your next release or iteration. A burndown chart that shows remaining issues and the rate at which they are being closed can help predict when all of the issues will be closed so you can accelerate issue resolution if necessary. (See "Managing requirements on agile projects" later in this chapter for a sample burndown chart.) Categorizing issues will help you determine which sections of requirements still need work. Few open issues on a section could mean either that the requirements haven't been reviewed yet or that the open issues are mostly resolved.

Nearly all of the defects logged early in a project are related to issues in the requirements, such as asking for clarification on a requirement, scope decisions, questions about development feasibility, and to-do items on the requirements themselves. All stakeholders can log questions as they review the requirements. Table 27-2 lists several common types of requirements issues that can arise.

TABLE 27-2 Common types of requirements issues

Issue type	Description
Requirement question	Something isn't understood or decided about a requirement.
Missing requirement	Developers uncovered a missed requirement during design or implementation.
Incorrect requirement	A requirement was wrong. It should be corrected or removed.
Implementation question	As developers implement requirements, they have questions about how something should work or about design alternatives.
Duplicate requirement	Two or more equivalent requirements are discovered. Delete all but one of them.
Unneeded requirement	A requirement simply isn't needed anymore.

Bad things can happen if you don't have an organized process for handling your requirements issues. On one project, a stakeholder mentioned very early on that we would handle something in "the portal." This was the first I had heard of a portal as part of the solution, so I asked about it. The stakeholder assured me that the COTS package being acquired included a portal component that simply had to be configured properly. We hadn't included any time for portal requirements in our plan, so I thought we might have a gap. I asked a teammate to record an issue about the portal so we wouldn't overlook that need. I left the project a few weeks later.

As it turned out, my teammate jotted the portal issue on a whiteboard that was later erased; she didn't record it in our issue-tracking tool. Six months into the project, our executive stakeholder came to me absolutely furious that no one had elicited requirements for the portal. I had to find out why we hadn't developed portal requirements: we simply forgot about it. Recording the issue in a tracking tool would have kept us from scrambling at the last minute and avoided upsetting the customer.

Measuring requirements effort

As with requirements development, your project plan should include tasks and resources for the requirements management activities described in this chapter. If you track how much effort you spend on requirements development and management activities, you can evaluate whether it was too little, about right, or too much, and adjust your future planning accordingly. Karl Wiegers (2006) discusses measuring various other aspects of the requirements work on a project.

Measuring effort requires a culture change and the individual discipline to record daily work activities (Wiegers 1996). Effort tracking isn't as time-consuming as people sometimes fear. Team members gain valuable insight from knowing how they *actually* spent their time, compared to how they *thought* they spent their time, compared to how they were *supposed* to spend their time. Effort tracking also indicates whether the team is performing the intended requirements-related activities.

Note that work effort is not the same as elapsed calendar time. Tasks can be interrupted; they might require interactions with other people that lead to delays. The total effort for a task, in units of labor hours, might not change because of such factors (although frequent interruptions do reduce an individual's productivity), but the calendar duration increases.

When tracking requirements development effort, you might find it valuable to separate the time spent by people in the BA role from time spent by other project participants. Tracking the BA's time will help you plan how much BA effort is needed on future projects (see Chapter 19, "Beyond requirements development," for more about estimating BA time). Measuring the total effort spent on requirements activities by all stakeholders gives you a sense of the total cost of requirements activities on a project. Record the number of hours spent on requirements development activities such as the following:

- Planning requirements-related activities for the project
- Holding workshops and interviews, analyzing documents, and performing other elicitation activities
- Writing requirements specifications, creating analysis models, and prioritizing requirements
- Creating and evaluating prototypes intended to assist with requirements development
- Reviewing requirements and performing other validation activities

Count the effort devoted to the following activities as requirements management effort:

- Configuring a requirements management tool for your project
- Submitting requirements changes and proposing new requirements
- Evaluating proposed changes, including performing impact analysis and making decisions
- Updating the requirements repository
- Communicating requirements changes to affected stakeholders
- Tracking and reporting requirements status
- Creating requirements trace information

Remember, the time you spend on these requirements-related activities is an investment in project success, not just a cost. To justify the activities, compare this time investment with the time the team spends dealing with issues that arose because these things were *not* done—the cost of poor quality.

Managing requirements on agile projects

Agile projects accommodate change by building the product through a series of development iterations and managing a dynamic product backlog of work remaining to be done. As described in Chapter 2, the stakeholders reach agreement on the stories to be implemented in each iteration. New stories that customers add while an iteration is under way are prioritized against the remaining

backlog contents and allocated to future iterations. New stories might displace lower-priority stories if the team wants to keep the original delivery schedule. The goal—as it should be for all projects—is to always be working on the highest-priority stories to deliver the maximum value to customers as quickly as possible. See Chapter 28 for more information about handling requirement changes on agile projects.

Some agile teams, particularly large or distributed teams, use an agile project management tool to track the status of an iteration and the stories allocated to it. The stories and their associated acceptance criteria and acceptance tests might all be placed in a product backlog or user story–management tool. Story status can be monitored by using statuses analogous to those described earlier in Table 27-1 (Leffingwell 2011):

- In backlog (the story is not yet allocated to an iteration)
- Defined (details of the story were discussed and understood, and acceptance tests were written)
- In progress (the story is being implemented)
- Completed (the story is fully implemented)
- Accepted (acceptance tests were passed)
- Blocked (the developer is unable to proceed until something else is resolved)

Agile projects typically monitor their progress with an iteration burndown chart (Cohn 2004; Cohn 2005). The team estimates the total amount of work to do on the project, often sized in units of story points, which are derived from an understanding of the user stories in the product backlog (Cohn 2005; Leffingwell 2011). The story point total is thus proportional to the amount of effort the team must expend to implement the requirements. The team allocates certain user stories to each iteration based on their priority and their estimated size in story points. The team's past or average velocity dictates the number of story points the team plans to deliver in an iteration of a particular calendar duration.

The team charts the story points remaining in the product backlog at the end of each iteration. This total will change as work is completed, as current stories are better understood and re-estimated, as new stories are added, and as customers remove pending work from the backlog. That is, rather than monitoring the count and status of individual functional requirements or features (which can come in a variety of sizes), the burndown chart shows the total work remaining to be done at a specific time.

Figure 27-3 illustrates a burndown chart for a hypothetical project. Notice that the scope remaining, as measured in story points, actually increased in iterations 2, 3, and 5. This indicates that more new functionality was added to the backlog than was completed or removed during the course of the iteration. The burndown chart helps the team avoid the "90 percent done" syndrome by making visible the amount of work remaining, as opposed to the amount of work completed, which doesn't reflect the inevitable scope increases. The slope of the burndown chart also reveals the projected end date for the project, the point at which no work remains in the backlog.

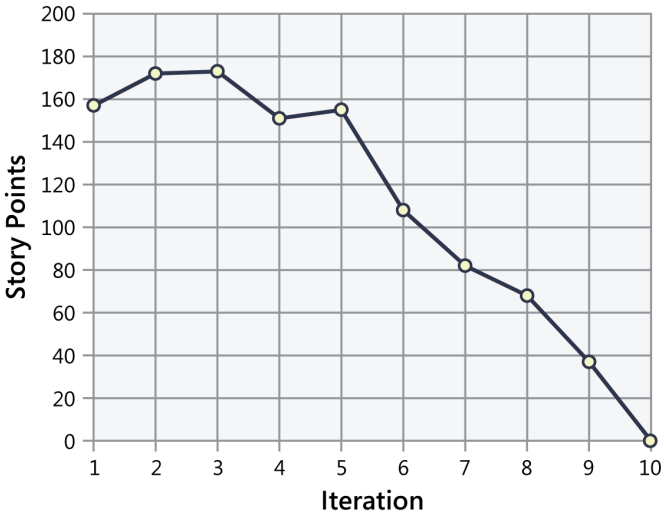

FIGURE 27-3 Sample iteration burndown chart for monitoring the product backlog on an agile project.

Why manage requirements?

Whether your project is following a sequential development life cycle, one of the various agile life cycles, or any other approach, managing the requirements is an essential activity. Requirements management helps to ensure that the effort you invest in requirements development isn't squandered. Effective requirements management reduces the expectation gap by keeping all project stakeholders informed about the current state of the requirements throughout the development process. It lets you know where you're headed, how the trip is going, and when you've arrived at your destination.

Next steps

- Document the processes your organization will follow to manage the requirements on each project. Engage several business analysts to draft, review, pilot, and approve the process activities and deliverables. The process steps you define must be practical and realistic, and they must add value to each affected project.
- If you're not using a requirements management tool, define a version labeling scheme to identify your requirements documents. Educate the BAs about this scheme.
- Select the statuses that you want to use to describe the life cycle of your functional requirements or user stories. Draw a state-transition diagram to show the conditions or events that trigger a change from one status to another.
- Define the current status for each requirement in your baseline. Keep the status current as development progresses.

CHAPTER 28

Change happens

"How's your development work coming, Glenn?" asked Dave, the Chemical Tracking System's project manager, during a status meeting.

"I'm not as far along as I'd planned to be," Glenn admitted. "I'm adding a new catalog query function for Harumi, and it's taking a lot longer than I expected."

Dave was puzzled. "I don't remember hearing about a new catalog query function. Did Harumi submit that request through the change process?"

"No, she approached me directly with the suggestion," said Glenn. "It seemed pretty simple, so I told her I'd work it in. It turned out not to be simple at all! Every time I think I'm done, I realize I missed a change needed in another file, so I have to fix that, rebuild the component, and test it again. I thought this would take about six hours, but I've spent almost three days on it so far. I know I'm holding up the next build. Should I finish adding this query function or go back to what I was working on before?"

Most developers have encountered an apparently simple change that turned out to be far more complicated than expected. Developers sometimes don't—or can't—produce realistic estimates of the cost and other ramifications of a proposed software change. Additionally, when developers who want to be accommodating agree to add enhancements that users request, requirements changes slip in through the back door instead of being approved by the right stakeholders. Such uncontrolled change is a common source of project chaos, schedule slips, quality problems, and hard feelings. This chapter describes both formal change control practices and how agile projects incorporate changes.

Why manage changes?

Software change isn't a bad thing; in fact, it's necessary. It's virtually impossible to define all of a product's requirements up front. The world changes as development progresses: new market opportunities arise, regulations and policies change, and business needs evolve. An effective software team can nimbly respond to necessary changes so that the product they build provides timely customer value. An organization that's serious about managing its software projects must ensure that:

- Proposed requirements changes are thoughtfully evaluated before being committed to.
- Appropriate individuals make informed business decisions about requested changes.

- Change activity is made visible to affected stakeholders.
- Approved changes are communicated to all affected participants.
- The project incorporates requirements changes in a consistent and effective fashion.

But change always has a price. Revising a simple webpage might be quick and easy; making a change in an integrated circuit design can cost tens of thousands of dollars. Even a rejected change request consumes the time needed to submit, evaluate, and decide to reject it. Unless project stakeholders manage changes during development, they won't really know what will be delivered, which ultimately leads to an expectation gap.

Problems can also arise if a developer implements a requirement change directly in the code without communicating with other team members. The documented requirements then become an inaccurate representation of what the product does. The code can become brittle if changes are made without respecting the architecture and design structure. On one project, developers introduced new and modified functionality that the rest of the team didn't discover until system testing. They didn't expect that functionality, and they didn't know how to test it. This required unplanned rework of test procedures and user documentation. Consistent change control practices help prevent such problems and the associated frustration, rework, and wasted time.

Beware subversive changes

A vendor and a customer once caused havoc when they bypassed the change process on a contracted project. The vendor (vetted by the IT department, but hired by the business area) was to develop a new mobile workstation application. Requirements were elicited collaboratively with 10 subject matter experts. Then the lead customer from the business area decided that she wanted more requirements changes. Not trusting that the revisions would be funded, she colluded with the vendor's developers to subvert the agreed-upon requirements. They rented a hotel room and worked in secret, making changes to the code on the fly. When testers found that the deliverable didn't match the requirements, the whole story came out. Backtracking the changes and expected outcomes cost the organization considerable time and effort.

By a strange twist of fate, that lead customer later became a business analyst. She took the time to apologize, because only then did she come to understand how her actions had undermined the rest of the team.

Managing scope creep

In an ideal world, you would document all of a new system's requirements before beginning construction, and they'd remain stable throughout the development effort. This is the premise behind the pure waterfall development model, but it doesn't work well in practice. At some point, you

must freeze the requirements for a specific release or development iteration or you'll never finish it. However, stifling change prematurely ignores the realities that customers aren't always sure what they need, business needs change, and developers want to respond to those changes.

Requirements growth includes new functionality and significant modifications that are presented after a set of requirements has been baselined (see Chapter 2, "Requirements from the customer's perspective"). The longer a project goes on, the more growth it experiences. The requirements for software systems typically grow between 1 percent and 3 percent per calendar month (Jones 2006). Some requirements evolution is legitimate, unavoidable, and even advantageous. Scope creep, though, in which the project continuously incorporates more functionality without adjusting resources, schedules, or quality goals, is insidious. The problem is not that requirements change but that late changes can have a big impact on work already performed. If every proposed change is approved, it might appear to stakeholders that the software will never be delivered—and indeed, it might not.

The first step in managing scope creep is to document the business objectives, product vision, project scope, and limitations of the new system, as described in Chapter 5, "Establishing the business requirements." Evaluate every proposed requirement or feature against the business requirements. Engaging customers in elicitation reduces the number of requirements that are overlooked. Prototyping helps to control scope creep by helping developers and users share a clear understanding of user needs and prospective solutions. Using short development cycles to release a system incrementally provides frequent opportunities for adjustments.

The most effective technique for controlling scope creep is the ability to say "no" (Weinberg 1995). People don't like to say "no," and development teams can receive intense pressure to always say "yes." Philosophies such as "the customer is always right" or "we will achieve total customer satisfaction" are fine in the abstract, but you pay a price for them. Ignoring the price doesn't alter the fact that change is not free. The president of one software tool vendor is accustomed to hearing the development manager say "not now" when he suggests a new feature. "Not now" is more palatable than a simple rejection. It holds the promise of including the feature in a subsequent release.

Trap Freezing the requirements for a new system too soon after initial elicitation activities is unwise and unrealistic. Instead, establish a baseline when you think a set of requirements is well enough defined for construction to begin, and then manage changes to minimize their adverse impact on the project.

Change control policy

Management should communicate a policy that states its expectations of how project teams will handle proposed changes in requirements and all other significant project artifacts. Policies are meaningful only if they are realistic, add value, and are enforced. The following change control policy statements can be helpful:

- All changes must follow the process. If a change request is not submitted in accordance with this process, it will not be considered.
- No design or implementation work other than feasibility exploration will be performed on unapproved changes.
- Simply requesting a change does not guarantee that it will be made. The project's change control board (CCB) will decide which changes to implement.
- The contents of the change database must be visible to all project stakeholders.
- Impact analysis must be performed for every change.
- Every incorporated change must be traceable to an approved change request.
- The rationale behind every approval or rejection of a change request must be recorded.

Of course, tiny changes will hardly affect the project, and big changes will have a significant impact. In practice, you might decide to leave certain requirements decisions to the developers' discretion, but no change affecting more than one individual's work should bypass your process. Include a "fast path" to expedite low-risk, low-investment change requests in a compressed decision cycle.

Basic concepts of the change control process

When performing a software process assessment, I asked a project team how they handled requirements changes. After an awkward silence, one person said, "Whenever the marketing representative wants to make a change, he asks Bruce or Robin because they always say 'yes.' The rest of us push back about changes." This didn't strike me as a great change process.

A sensible change control process lets the project's leaders make informed business decisions that will provide the greatest customer and business value while controlling the product's life-cycle cost and the project's schedule. The process lets you track the status of all proposed changes, and it helps ensure that suggested changes aren't lost or overlooked. After you've baselined a set of requirements, you should follow this process for all proposed changes to that baseline.

Stakeholders sometimes balk at being asked to follow a new process, but a change control process is not an obstacle to making necessary modifications. It's a funneling and filtering mechanism to

ensure that the project expeditiously incorporates the most appropriate changes. If a proposed change isn't important enough for a stakeholder to take just a couple of minutes to submit it through a standard, simple channel, then it's not worth considering for inclusion. Your change process should be well documented, as simple as possible, and—above all—effective.

> **Trap** If you ask your stakeholders to follow a new change control process that's ineffective, cumbersome, or too complicated, people will find ways to bypass the process—and they should.

Managing requirements changes is similar to the process for collecting and making decisions about defect reports. The same tools can support both activities. Remember, though: a tool is not a substitute for a documented process, and neither one is a substitute for appropriate discussions between stakeholders. Regard both a tool and a written process as ways to support these critical conversations.

When you need to incorporate a change, start at the highest level of abstraction that the change touches and cascade the change through affected system components. For example, a proposed change might affect a user requirement but not any business requirements. Modifying a high-level system requirement could affect numerous software and hardware requirements in multiple subsystems. Some changes pertain only to system internals, such as the way a communication service is implemented. These aren't user-visible requirements changes, but rather design or code changes.

A change control process description

Figure 28-1 illustrates a template for a change control process description to handle requirements modifications. A sample change control process description is available for downloading from this book's companion content website. If this template is too elaborate for your environment, scale it down for more informal projects. We find it helpful to include the following four components in all process descriptions:

- Entry criteria, the conditions that must be satisfied before the process execution can begin
- The various tasks involved in the process, the project role responsible for each task, and other participants in the task
- Steps to verify that the tasks were completed correctly
- Exit criteria, the conditions that indicate when the process is successfully completed

The rest of this section describes the various sections in the change control process description.

1. **Purpose and scope**
2. **Roles and responsibilities**
3. **Change request states**
4. **Entry criteria**
5. **Tasks**
   - 5.1 Evaluate change request
   - 5.2 Make change decision
   - 5.3 Implement the change
   - 5.4 Verify the change
6. **Exit criteria**
7. **Change control status reporting**

Appendix: Attributes stored for each request

FIGURE 28-1 Sample template for a change control process description.

1. Purpose and scope

Describe the purpose of this process and the organizational scope to which it applies. Indicate whether any specific kinds of changes are exempted, such as changes in interim work products. Define any terms that are necessary for understanding the rest of the document.

2. Roles and responsibilities

List the project team roles that participate in the change control activities and describe their responsibilities. Table 28-1 suggests some pertinent roles; adapt these to each project situation. Different individuals need not be required for each role. For example, the CCB Chair might also receive submitted change requests. The same person can fill several—perhaps all—roles on a small project. As one experienced project manager put it, "What I find important is that the representation of the CCB needs to be able to speak to the needs of the diverse stakeholders, including the end users, the business, and the development community: do we need it, can we sell it, can we build it?"

TABLE 28-1 Possible project roles in change-management activities

Role	Description and responsibilities
CCB Chair	Chairperson of the change control board; generally has final decision-making authority if the CCB does not reach agreement; identifies the Evaluator and the Modifier for each change request
CCB	The group that decides to approve or reject proposed changes for a specific project
Evaluator	Person whom the CCB Chair asks to analyze the impact of a proposed change
Modifier	Person who is responsible for making changes in a work product in response to an approved change request
Originator	Person who submits a new change request
Request Receiver	Person who initially receives newly submitted change requests
Verifier	Person who determines whether the change was made correctly

3. Change request status

A change request passes through a defined life cycle of states. You can represent these states by using a state-transition diagram (see Chapter 12," A picture is worth 1024 words"), as illustrated in Figure 28-2. Update a request's status only when the specified transition criteria are met. For instance, you can set the state to "Change Made" after all affected work products have been modified to implement the change, whether that is just a single requirement statement or a set of related development work products.

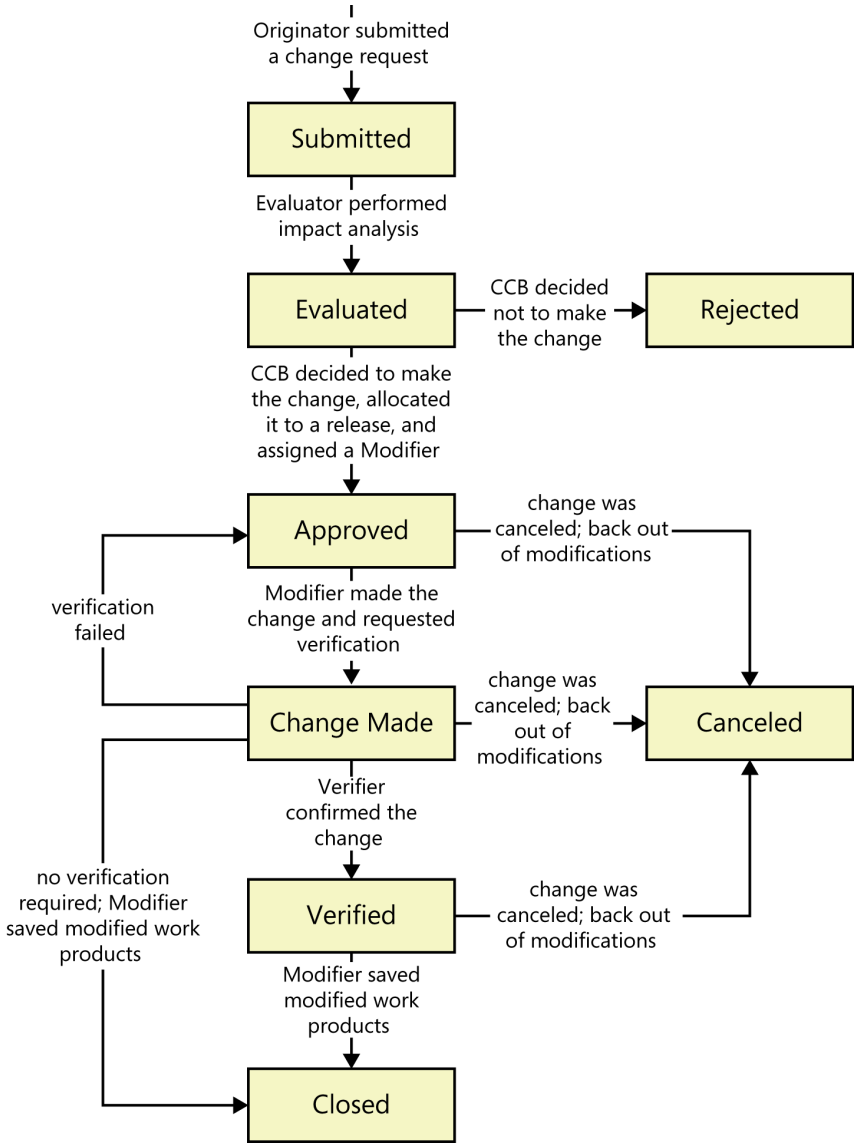

FIGURE 28-2 State-transition diagram for a change request.

4. Entry criteria

The basic entry criterion for your change control process is that a change request with all the necessary information has been received through an approved channel. All potential originators should know how to submit a change request. Your change tool should assign a unique identifier to each request and route all changes to the Request Receiver.

5. Tasks

This section of the process describes the tasks that are performed to handle a single change request.

5.1 Evaluate change request

Begin by evaluating the request for technical feasibility, cost, and alignment with the project's business requirements and resource constraints. The CCB Chair might assign an Evaluator to perform impact analysis, risk and hazard analysis, or other assessments. (See the "Change impact analysis" section later in this chapter.) This ensures that the consequences of accepting the change are understood. The Evaluator and the CCB should also consider the business and technical implications, if any, of rejecting the request.

5.2 Make change decision

The appropriate decision makers, chartered as the CCB, then decide whether to approve or reject the change. The CCB gives each approved change a priority or target implementation date, or it allocates the change to a specific iteration or release. It might simply add a new requirement to the product backlog of pending work. The CCB updates the request's status and notifies all affected team members.

5.3 Implement the change

The assigned Modifier (or Modifiers) updates the affected work products as necessary to fully implement the change. Use requirements trace information to find all the parts of the system that the change touches, and revise the trace information if necessary to reflect the changes made.

5.4 Verify the change

Requirements changes typically are verified through a peer review to ensure that modified deliverables correctly address all aspects of the change. Multiple team members might verify the changes made in various downstream work products through testing or review. After verification is complete, the Modifier stores updated work products in the appropriate locations per the project's document and code management conventions.

6. Exit criteria

Satisfying the following exit criteria indicates that an execution of your change control process was properly completed:

- ❑ The status of the request is Rejected, Closed, or Canceled.
- ❑ All modified work products are updated and stored in the correct locations.
- ❑ The relevant stakeholders have been notified of the change details and the status of the change request.

7. Change control status reporting

Identify the charts and reports you'll use to summarize the contents of the change database. These charts might show the number of change requests in each state as a function of time, or trends in the average time that a change request is unresolved. Describe the procedures for producing the charts and reports. The project manager uses these reports when tracking the project's status.

Appendix: Attributes stored for each request

Table 28-2 lists some data attributes to consider storing for each change request. Some of these items are supplied by the Originator and some by the CCB. In your change control process, indicate which attributes are required and which are optional. Don't define more attributes than you really need. Your change tool should handle some of these (ID, date submitted, date updated) automatically.

TABLE 28-2 Suggested change request attributes

Item	Description
Change origin	Functional area that requested the change; possible groups include marketing, management, customer, development, and testing
Change request ID	Unique identifier assigned to the request
Change type	Type of change request, such as requirement change, proposed enhancement, or defect report
Date submitted	Date the Originator submitted the change request
Date updated	Date the change request was most recently modified
Description	Free-form text description of the change being requested
Implementation priority	The relative importance of making the change as determined by the CCB: low, medium, or high
Modifier	Person who is primarily responsible for implementing the change
Originator	Person who submitted this change request
Originator priority	The relative importance of making the change from the Originator's point of view: low, medium, or high
Planned release	Product release or iteration for which an approved change is scheduled
Project	Name of the project in which a change is being requested

Item	Description
Response	Free-form text of responses made to the change request; multiple responses can be made over time; do not change existing responses when entering a new one
Status	The current status of the change request, selected from the options in Figure 28-2
Title	One-line summary of the proposed change
Verifier	Person who is responsible for determining whether the change was made correctly

The change control board

The *change control board* is the body of people—whether it is one individual or a diverse group—that decides which proposed changes and new requirements to accept, which to accept with revisions, and which to reject. The CCB also decides which reported defects to correct and when to correct them. Some CCBs are empowered to make decisions, whereas others can only make recommendations for management decision. Projects always have some de facto group that makes change decisions. Establishing a CCB formalizes this group's composition and authority and defines its operating procedures.

To some people, the term "change control board" conjures an image of wasteful bureaucratic overhead. Instead, think of the CCB as providing a valuable structure to help manage even a small project. On a small project, it makes sense to have only one or two people make the change decisions. Very large projects or programs might have several levels of CCBs, some responsible for business decisions, such as requirements changes, and some for technical changes. A large program that encompasses multiple projects would establish a program-level CCB and an individual CCB for each project. Each project CCB resolves issues and changes that affect only that project. Issues that affect multiple projects and changes that exceed a specified cost or schedule impact are escalated to the program-level CCB.

CCB composition

The CCB membership should represent all groups who need to participate in making decisions within the scope of that CCB's authority. Consider selecting representatives from the following areas:

- Project or program management
- Business analysis or product management
- Development
- Testing or quality assurance
- Marketing, the business for which the application is being built, or customer representatives
- Technical support or help desk

Only the subset of these people who need to make the decisions will be part of the CCB, although all stakeholders must be informed of decisions that affect their work. The CCB for a project with both

software and hardware components might also include representatives from hardware engineering, systems engineering, and/or manufacturing. Keep the CCB small so the group can respond promptly and efficiently to change requests. Make sure the CCB members understand and accept their responsibilities. Invite other individuals to CCB meetings as necessary to ensure that the group has adequate technical and business information.

CCB charter

All of the project teams in an organization can follow the same change control process. However, their CCBs might function in different ways. Each project should create a brief charter (which could be part of the project management plan) that describes its CCB's purpose, scope of authority, membership, operating procedures, and decision-making process (Sorensen 1999). A template for a CCB charter is available for downloading from this book's companion content website. The charter should state the frequency of regularly scheduled CCB meetings and the conditions that trigger a special meeting or decision. The scope of the CCB's authority indicates which decisions it can make and which ones it must escalate.

Making decisions

Each CCB needs to define its decision-making process, which should indicate:

- The number of CCB members or the key roles that constitute a decision-making quorum.
- The decision rules to be used (see Chapter 2 for more about decision rules).
- Whether the CCB Chair can overrule the CCB's collective decision.
- Whether a higher level of CCB or management must ratify the group's decision.

The CCB balances the anticipated benefits against the estimated impact of accepting a proposed change. Benefits from improving the product could include financial savings, increased revenue, higher customer satisfaction, and competitive advantage. Possible negative impacts include increased development and support costs, delayed delivery, and degraded product quality.

> **Trap** Because people don't like to say "no," it's easy to accumulate a huge backlog of approved change requests that will never get done. Before accepting a proposed change, make sure you understand the rationale behind it and the business value the change will provide.

Communicating status

After the CCB makes its decision, a designated individual updates the request's status in the change database. Some tools automatically generate an email message to communicate the new status to the Originator who proposed the change and to others affected by the change. If an email message is not generated automatically, inform the affected people so they can respond to the change.

Renegotiating commitments

Stakeholders can't stuff more and more functionality into a project that has schedule, staff, budget, or quality constraints and still expect to succeed. Before accepting a significant requirement change, renegotiate commitments with management and customers to accommodate the change. You might ask for more time or to defer lower-priority requirements. If you don't obtain some commitment adjustments, document the threats to success in your project's risk list so people aren't surprised if there are negative outcomes.

Change control tools

Many teams use commercial issue-tracking tools to collect, store, and manage requirements changes. A report of recently submitted change requests extracted from the tool can serve as the agenda for a CCB meeting. Issue-tracking tools can report the number of requests having each state at any given time. Because the available tools, their vendors, and their features frequently change, we don't provide specific tool recommendations here. To support your change process, look for a tool that:

- Allows you to define the attributes that constitute a change request.
- Allows you to implement a change request life cycle with multiple change request statuses.
- Enforces the state-transition model so that only authorized users can make specific status changes.
- Records the date of each status change and the identity of the person who made it.
- Provides customizable, automatic email notification when an Originator submits a new request or when a request's status is updated.
- Produces both standard and custom reports and charts.

Some commercial requirements management tools have a change-request system built in. These systems can link a proposed change to a specific requirement so that the individual responsible for each requirement is notified by email whenever someone submits a pertinent change request.

Tooling up a process

When I worked on a web development team, one of our first process improvements was to implement a change control process to manage our huge backlog of change requests (Wiegers 1999). We began with a process like the one described in this chapter. We piloted it for a few weeks by using paper forms while I evaluated several issue-tracking tools. During the pilot process we discovered ways to improve the process and additional data attributes for the change requests. We selected a highly configurable tool and tailored it to match our process. The team used this process and tool to handle requirements changes in systems under development, defect reports and suggested enhancements for production systems, and requests for new projects. Change control was one of our most successful process improvement initiatives.

Measuring change activity

Measuring change activity is a way to assess the stability of the requirements. It also reveals opportunities for process improvements that might lead to fewer changes in the future. Consider tracking the following aspects of your requirements change activity:

- The total number of change requests received, currently open, and closed
- The cumulative number of added, deleted, and modified requirements
- The number of requests that originated from each change origin
- The number of changes received against each requirement since it was baselined
- The total effort devoted to processing and implementing change requests

You don't necessarily need to monitor your requirements change activities to this degree. As with all software metrics, understand your goals and how you'll use the data before you decide what to measure (Wiegers 2007). Start with simple metrics to begin establishing a measurement culture in your organization and to collect the data you need to manage your projects effectively.

Figure 28-3 illustrates a way to track the amount of requirements change your project experiences during development (Wiegers 2006). This requirements volatility chart tracks the rate at which new proposals for requirements changes arrive after a baseline was established. This chart should trend toward zero as you approach release. A sustained high frequency of changes implies a risk of failing to meet your schedule commitments. It probably also indicates that the original requirements set was incomplete; better elicitation practices might be in order.

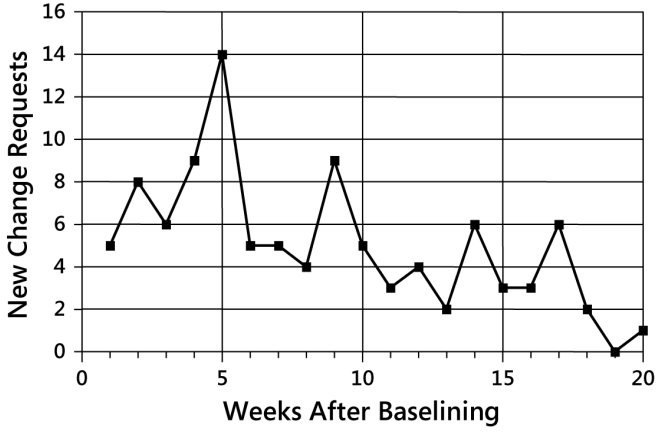

FIGURE 28-3 Sample chart of requirements change activity.

Tracking the requirements change origins is also illuminating. Figure 28-4 shows a way to represent the number of change requests that came from different sources. The project manager could discuss a chart like this with the marketing manager and point out that marketing has requested the most requirements changes. This might lead to a fruitful discussion about actions the team could take to

reduce the number of changes received from marketing in the future or better ways to handle them. Using data as a starting point for such discussions is more constructive than holding a confrontational debate fueled by emotion. Come up with your own list of possible requirements change origins.

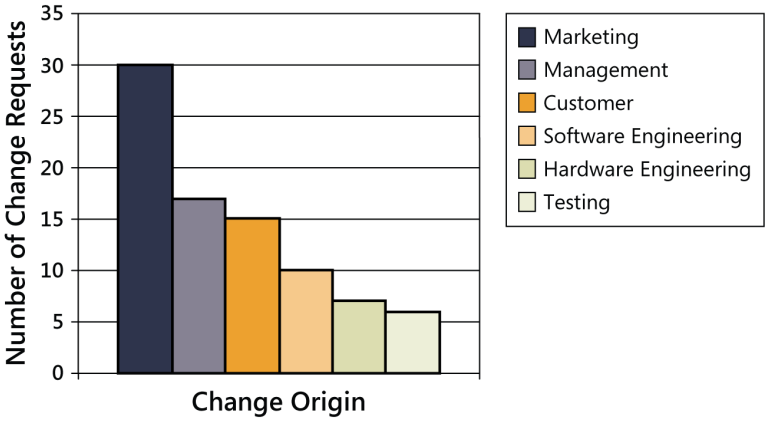

FIGURE 28-4 Sample chart of requirement change origins.

Change impact analysis

The need for impact analysis is obvious for major enhancements. However, unexpected complications can lurk below the surface of even minor change requests. A company once had to change the text of one error message in its product. What could be simpler? The product was available in both English-language and German-language versions. There were no problems in English, but in German the new message exceeded the maximum character length allocated for error message displays in both the message box and a database. Coping with this seemingly simple change request turned out to be much more work than the developer had anticipated when he promised a quick turnaround.

Impact analysis is a key aspect of responsible requirements management (Arnold and Bohner 1996). It provides an accurate understanding of the implications of a proposed change, which helps the team make informed business decisions about which proposals to approve. The analysis examines the request to identify components that might have to be created, modified, or discarded, and to estimate the effort required to implement the change. Before a developer says, "Sure, no problem" in response to a change request, he should spend a little time on impact analysis.

Impact analysis procedure

The CCB Chair will ask one or more technical people (business analysts, developers, and/or testers) to perform the impact analysis for a specific change proposal. Impact analysis involves three steps:

1. Understand the possible implications of making the change. A requirement change often produces a large ripple effect, leading to modifications in other requirements, architectures,

designs, code, and tests. Changes can lead to conflicts with other requirements or can compromise quality attributes, such as performance or security.

2. Identify all the requirements, files, models, and documents that might have to be modified if the team incorporates the requested change.
3. Identify the tasks required to implement the change, and estimate the effort needed to complete those tasks.

Important Skipping impact analysis doesn't change the size of the task. It just turns the size into a surprise. Software surprises are rarely good news.

Figure 28-5 presents a checklist of questions to help the evaluator understand the implications of accepting a proposed change. The checklist in Figure 28-6 contains questions to help identify all software elements and other work products that the change might affect. Requirements trace information that links the affected requirement to other downstream deliverables helps greatly with impact analysis. As you gain experience in using these checklists, modify them to suit your own projects. (Note: Figures 28-5 through 28-8 are available for downloading from this book's companion content website.)

- ❑ Will the change enhance or impair the ability to satisfy any business requirements?
- ❑ Do any existing requirements in the baseline conflict with the proposed change?
- ❑ Do any other pending requirements changes conflict with the proposed change?
- ❑ What are the business or technical consequences of not making the change?
- ❑ What are possible adverse side effects or other risks of making the proposed change?
- ❑ Will the proposed change adversely affect performance or other quality attributes?
- ❑ Is the proposed change feasible within known technical constraints and current staff skills?
- ❑ Will the proposed change place unacceptable demands on any resources required for the development, test, or operating environments?
- ❑ Must any tools be acquired to implement and test the change?
- ❑ How will the proposed change affect the sequence, dependencies, effort, or duration of any tasks currently in the project plan?
- ❑ Will prototyping or other user input be required to validate the change?
- ❑ How much effort that has already been invested in the project will be lost if this change is accepted?
- ❑ Will the proposed change cause an increase in product unit cost, such as by increasing third-party product licensing fees?
- ❑ Will the change affect any marketing, manufacturing, training, or customer support plans?

FIGURE 28-5 Questions to understand the possible implications of a proposed change.

- ❑ Identify any user interface changes, additions, or deletions required.
- ❑ Identify any changes, additions, or deletions required in reports, databases, or files.
- ❑ Identify the design components that must be created, modified, or deleted.
- ❑ Identify the source code files that must be created, modified, or deleted.
- ❑ Identify any changes required in build files or procedures.
- ❑ Identify existing unit, integration, and system tests to be modified or deleted.
- ❑ Estimate the number of new unit, integration, and system tests needed.
- ❑ Identify help screens, training or support materials, or other user documentation that must be created or modified.
- ❑ Identify other applications, libraries, or hardware components affected by the change.
- ❑ Identify any third-party software to be acquired or modified.
- ❑ Identify any impact the proposed change will have on the project management plan, quality assurance plan, configuration management plan, or other plans.

FIGURE 28-6 Checklist to determine work products that might be affected by a proposed change.

Many estimation problems arise because the estimator doesn't think of all the work required to complete an activity. Therefore, this impact analysis approach emphasizes thorough task identification. For substantial changes, use a small team—not just one developer—to do the analysis and effort estimation to avoid overlooking important tasks. Following is a simple procedure for evaluating the impact of a proposed requirement change:

1. Work through the checklist in Figure 28-5.
2. Work through the checklist in Figure 28-6. Some requirements management tools include an impact analysis report that follows traceability links and finds the system elements that depend on the requirements affected by a change request.
3. Use the worksheet in Figure 28-7 to estimate the effort required for the anticipated tasks. Most change requests will require only a portion of the tasks on the worksheet.
4. Sum the effort estimates.
5. Identify the sequence in which the tasks must be performed and how they can be interleaved with currently planned tasks.
6. Estimate the impact of the proposed change on the project's schedule and cost.
7. Evaluate the change's priority compared to other pending requirements.
8. Report the impact analysis results to the CCB.

Hours	Task
_____	Update the SRS or requirements repository
_____	Develop and evaluate a prototype
_____	Create new design components
_____	Modify existing design components
_____	Develop new user interface components
_____	Modify existing user interface components
_____	Develop new user documentation and help screens
_____	Modify existing user documentation and help screens
_____	Develop new source code
_____	Modify existing source code
_____	License and integrate third-party software
_____	Modify build files and procedures
_____	Write new unit and integration tests
_____	Modify existing unit and integration tests
_____	Perform unit and integration testing after implementation
_____	Write new system and acceptance tests
_____	Modify existing system and acceptance tests
_____	Modify automated test suites
_____	Perform regression testing
_____	Develop new reports
_____	Modify existing reports
_____	Develop new database elements
_____	Modify existing database elements
_____	Develop new data files
_____	Modify existing data files
_____	Modify various project plans
_____	Update other documentation
_____	Update the requirements traceability matrix
_____	Review modified work products
_____	Perform rework following reviews and testing
_____	Other tasks
_____	**Total Estimated Effort**

FIGURE 28-7 Worksheet for estimating effort of a requirement change.

In most cases, this procedure shouldn't take more than a couple of hours to complete for a single change request. This seems like a lot of time to a busy developer, but it's a small investment in making sure the project wisely invests its limited resources. To improve your future impact analysis, compare the actual effort needed to implement each change with the estimated effort. Understand the reasons for any differences, and modify the impact estimation checklists and worksheet to help ensure that future impact analyses are more accurate.

Money down the drain

Two developers at the A. Datum Corporation estimated that it would take four weeks to add an enhancement to one of their information systems. The customer approved the estimate, and the developers set to work. After two months, the enhancement was only about half done and the customer lost patience: "If I'd known how long this was really going to take and how much it was going to cost, I wouldn't have approved it. Let's forget the whole thing." In the rush to begin implementation, the developers didn't do enough impact analysis to develop a reliable estimate that would let the customer make a good business decision. Consequently, the company wasted several hundred hours of work that could have been avoided with a few hours of impact analysis.

Impact analysis template

Figure 28-8 suggests a template for reporting the results from analyzing the impact of a requirement change. The people who will implement the change will need the analysis details and the effort planning worksheet, but the CCB needs only the summary of analysis results. As with all templates, try it and then adjust it to meet your project needs.

Change request ID: _____
Title: _____
Description: _____
_____
Evaluator: _____
Date prepared: _____
Estimated total effort: _____ labor hours
Estimated schedule impact: _____ days
Additional cost impact: _____ dollars
Quality impact: _____
_____
Other components affected: _____
_____
Other tasks affected: _____
Life-cycle cost issues: _____

FIGURE 28-8 Impact analysis template.

Change management on agile projects

Agile projects are specifically structured to respond to—and even welcome—scope changes. One of the 12 principles of agile software development is "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage" (*www.agilemanifesto.org/principles.html*). This principle acknowledges the reality that requirements

changes are inevitable, necessary, and often valuable. Accepting change helps to meet evolving business objectives and priorities and to accommodate the limitations of human plans and foresight.

Agile projects manage change by maintaining a dynamic backlog of work to be done (see Figure 28-9). "Work" includes user stories yet to be implemented, defects to be corrected, business process changes to be addressed, training to be developed and delivered, and the myriad other activities involved with any software project. Each iteration implements the set of work items in the backlog that have the highest priority at that time. As stakeholders request new work, it goes into the backlog and is prioritized against the other backlog contents. Work that has not yet been allocated can be reprioritized or removed from the backlog at any time. A new, high-priority story could be allocated to the forthcoming iteration, forcing a lower-priority story of about the same size to be deferred to a later iteration. Carefully managing the scope of each iteration ensures that it is completed on time and with high quality.

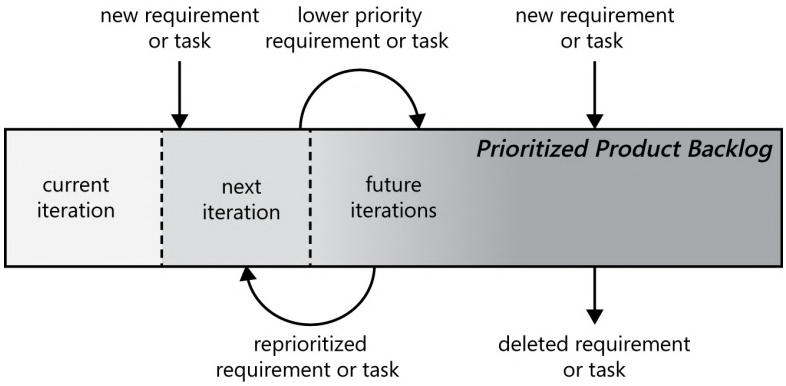

FIGURE 28-9 Agile projects manage change with a dynamic product backlog.

Because of the iterative nature of agile projects, every few weeks there will be an opportunity to select a set of work items from the backlog for the next development iteration. Agile teams vary as to whether new work that arrives during an iteration is always deferred to a future iteration, or whether they can modify the contents of the current iteration. Keeping the contents of an iteration frozen while it is under way provides stability for developers and predictability regarding what stakeholders can expect out of the iteration. On the other hand, adjusting the iteration's contents makes the team more responsive to customer needs.

Agile methods vary as to their philosophy on this point; there is no single "correct" approach. Either freeze the baseline for an iteration once it is under way or introduce high-priority changes as soon as you learn about them, whatever you think will work best for your team and the project's business objectives. The basic principle is to avoid both excessive change (churning requirements) and excessive rigidity (frozen requirements) within an iteration. One solution is to set the iteration length to the right duration for keeping most change out of the current iteration. That is, if changes need to be introduced too often, the standard iteration length might need to be shortened.

All agile methods define a role representing the end-user and customer constituencies. In Scrum this is the product owner role; in Extreme Programming this is the customer role. The customer or product owner has primary responsibility for prioritizing the contents of the product backlog.

He also makes decisions to accept proposed requirements changes, based on their alignment with the overarching product vision and the business value they will enable (Cohn 2010).

Because an agile team is a collaborative and cross-functional group of developers, testers, a business analyst, a project manager, and others, the team is already configured like the change control board discussed earlier in the chapter. The short duration of agile iterations and the small increment of product delivered in each iteration allows agile teams to perform change control frequently but on a limited scale. However, even agile projects must evaluate the potential cost of changes in requirements and their impact on product components. Scope changes that could affect the overall cost or duration of the project need to be escalated to a higher-level change authority, such as the project sponsor (Thomas 2008).

No matter what kind of project you're working on or what development life cycle your team is following, change is going to happen. You need to expect it and be prepared to handle it. Disciplined change-management practices can reduce the disruption that changes can cause. The purpose of change control is not to inhibit change, nor to inhibit stakeholders from proposing changes. It is to provide visibility into change activity and mechanisms by which the right people can consider proposed changes and incorporate appropriate ones into the project at the right time. This will maximize the business value and minimize the negative impact of changes on the team.

Next steps

- Identify the decision makers on your project, and set them up as a change control board. Have the CCB adopt a charter to establish and document the board's purpose, composition, and decision-making process.
- Define a state-transition diagram for the life cycle of proposed requirements changes in your project, starting with the diagram in Figure 28-2. Write a process to describe how your team will handle proposed requirements changes. Use the process manually until you're convinced that it's practical and effective.
- Select an issue-tracking tool that's compatible with your development environment. Tailor it to align with the process you created in the previous step.
- The next time you evaluate a requirement change request, first estimate the effort using your old method. Then estimate it again using the impact analysis approach described in this chapter. If you implement the change, compare the two estimates to see which agrees more closely with the actual effort required. Modify the impact analysis checklists and worksheet based on your experience to improve their future value.