SUCCEEDING WITH AGILE

Software Development Using Scrum

MIKE COHN

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Chapter 13

The Product Backlog

The biggest question looming at the start of a project is, what exactly are we building? We know the general shape of the system to be built. We may know, for example, that we are building a word processor. But there are always dark corners yet to be explored or issues yet to be settled about how specific features will work. Will our word processor include an interactive table design feature, or will tables be designed by entering values into a series of screens?

When using a sequential development process, we try to start with a lengthy, up-front requirements-gathering phase during which the product is presumably fully specified. The idea is that, by thinking longer, harder, and better at the outset of the project, no dark corners will be encountered during the main development phase of the project.

A Scrum team forgoes a lengthy, up-front requirements phase in favor of a just-in-time approach. High-level feature descriptions may be gathered early, but they are minimally described at that time and are progressively refined as the project progresses. They are documented in a *product backlog*, which is a list of all desired functionality not yet in the product. It is maintained by the product owner and kept in priority order, which is why a product backlog is sometimes called a *prioritized feature list*. Unlike a traditional requirements document, a product backlog is highly dynamic; items are added, removed, and reprioritized each sprint as more is learned about the product, the users, the team, and so on.

In this chapter we look at three changes organizations need to make to effectively work with a product backlog. First, we look at the need to shift from writing about a product's features to talking about them. Second, we see why it's important for detail to be added progressively rather than for all of it to be documented up front. Third, we see why specification by example should be a team's preferred approach to documenting a product's functionality. The chapter concludes with an acronym for remembering key attributes of the product backlog.

Shift from Documents to Discussions

There is a grand myth about requirements—if you write them down, users will get exactly what they want. That's not true. At best, users will get exactly what was written down, which may or may not be anything like what they really want. Written words are misleading—they look more precise than they are. For example, recently I wanted to run a three-day public training course. My assistant and I had discussed this, so I sent her an e-mail saying, "Please book the Hyatt in Denver," and reminded her of the dates. The next day she e-mailed me, "The hotel is booked." I e-mailed back, "Thanks," and turned my attention toward other matters.

About a week later she e-mailed me saying, "The hotel is booked on the days you wanted. What do you want to me do? Do you want to try another hotel in Denver? A different week? A different city?" She and I had completely miscommunicated about the meaning of "booked." When she told me "the hotel is booked," she meant, "The room we usually use at the Hyatt is already taken." When I read "the hotel is booked," I took it as a confirmation that she had booked the hotel like I had requested. Neither of us did anything wrong in this exchange. Rather, it is an example of how easy it is to miscommunicate, especially with written language. If we had been talking rather than e-mailing, I would have thanked her when she told me "the hotel is booked." The happy tone of my voice would have confused her, and we would have caught our miscommunication right then.

Beyond this problem, there are other reasons to favor discussions over documents.

Written documents can make you suspend judgment. When something is written, it looks official, formal, and finished, especially when fancy formatting has been applied. Awhile back a client whose office I'd visited many times decided we would have an off-site meeting near the company's office. The client sent me very detailed directions from my hotel to a country club where we were to meet:

- Turn left out of your hotel onto North Commerce Parkway and go 0.4 miles
- Turn left on SW 106th Avenue and go 0.2 miles
- Turn right on Royal Palm Boulevard for 1.1 miles
- Turn left onto Town Center

But I couldn't turn left onto Town Center! After 1.1 miles on Royal Palm, I found myself at an intersection, but Town Center went only to the right. I had been told to turn left but that road was called Weston Hills Boulevard. I could see the Country Club to the left, and it seemed like I should turn there. However, the directions had been very specific and correct to this point so I continued

forward. I went another two miles, watching the country club fade past me on the left. Eventually it was clear that the one instruction had been wrong, so I turned around and turned on Weston Hills instead of Town Center as was written in the directions. Suppose instead of these directions my client had simply said, "Head toward our office the same way you usually do. But when you see the country club, turn left. I don't know the name of that street, but you can't miss the country club."

With a written document, we don't iterate over meaning as we would in conversation. A few years back I read a requirements document that described a Windows Explorer–like interface for managing folders of data. One requirement said, "The name of the folder can be 127 characters." I was fairly certain that the requirement should have said that the folder name could be a *maximum of* 127 characters. But this was a bioinformatics application, and there were some unusual requirements such as text fields that could contain only the letters A, C, G, and T. A folder name of exactly 127 characters was a little surprising, but it was not impossible to fathom for this particular application.

Because a specific length was given, I presumed it must have been chosen for a good reason. It may not have been. Yet the nature of a requirements document made me much less likely to question the "127" mandate than I would have been had the analyst and I been talking. If we had, our conversation would have been punctuated with exchanges such as, "So what you're saying is…," "If I understand you, that means…," and "Doesn't that imply…." These questions are intended to ensure that a transfer of understanding has occurred, that I understand what you've said. This iterating over meaning is missing in documents.

Written documents decrease whole-team responsibility. One of the goals of shifting to Scrum is to get the whole team working together toward the goal of delivering a great product. We want to strip our development process of bad habits that work against this goal. Written documents create sequential hand-offs, which deprive the team of a unity of purpose. One person (or group) defines the product; another group builds it. Two-way communication is discouraged. Through the written document, one team member is saying, "Here's what to do," and others are expected to do it. This type of master-and-servant relationship is unlikely to create strong feelings of engagement on the part of the servants. Rather than feeling responsible for the success of the product, they feel responsible for doing what is described in the document. Discussions have the opposite effect: Whole team discussions lead to greater buy-in by all team members.

SEE ALSO

Whole-team responsibility and the problems with hand-offs were covered in Chapter 11, "Teamwork."

OBJECTION

"I can't get rid of all documents—my project has ISO 9001 (or similar) requirements, and everything has to be documented and traceable."

As I'll describe in the next section, you don't need to get rid of all documents. Eliminate those you can and keep others as short as possible, even considering whether they can be automatically generated. It is also important to recognize that you can document for posterity, while still relying on conversation during the project.

Don't Throw the Baby Out with the Documentation

These weaknesses of written communication are not to say we should abandon written requirements documents—absolutely not. Rather, we should use documents where appropriate. Because the Agile Manifesto says that we favor "working software over comprehensive documentation" (Beck et al. 2001), agile has been misinterpreted as being against documentation. The goal in agile development is to find the right balance between documentation and discussion. In the past we've often been skewed way too far toward the side of documents.

SEE ALSO

The section "Learn to Start Without a Specification," later in this chapter will show the power of specifying behavior in test cases.

We should also remain aware that requirements documents are just one form of documentation that may exist on a project. Other artifacts will exist: Test plans, executable test cases, and even code document the behavior (or intended behavior) of the system.

Because code and automated test cases will be produced to deliver a product, an experienced Scrum team learns to lean heavily on these artifacts. It will augment these forms of documentation with a written requirements document to the extent that such a document is helpful or required for regulatory, contractual, or legal purposes. A written requirements document will still be useful on many projects. Tom Poppendieck, coauthor of books on lean software development, has said that "when documents are mostly to enable handoffs, they are evil. When they capture a record of a conversation that is best not forgotten, they are valuable."

Use User Stories for the Product Backlog

User stories are the best way to shift the focus from writing about features to talking about them. A user story is a short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. User stories are often written on index cards or sticky notes, stored in a shoe box, and arranged on walls or tables to facilitate planning

and discussion. As such, user stories strongly shift the focus from writing about features to discussing them. User stories typically follow a simple template:

As a <type of user>, I want <some goal> so that <some reason>.

NOTE

Other templates are possible. The following template, for example, is promoted as putting the value of the user story at the front: *In order to <achieve value>, as <type of user>, I want <some goal>*. Having used both formats, I still prefer starting with *as a <type of user>*. For reasons why, see http://blog.mountaingoatsoftware.com/advantages-of-the-as-a-user-i-want-user-story-template. More important than the format of the written part of the user story, though, is that the conversations surrounding the story occur.

User stories can be stored in a software tool (and there are many reasons why you may choose to do so), but whenever possible I prefer to write on simple 3" × 5" index cards. Although a user story is often written on an index card or sticky note, the text written there is only the beginning. The story card is not meant to be a complete feature description in the same way we would view "The system shall…" statements in a software requirements specification. Instead, the story card serves as a two-way promise between the development team and the product owner. Team members promise they will talk to the product owner before beginning work on the story; the product owner promises to be available when the team is ready to talk.

The team's promise to talk to the product owner before beginning work is important because it frees the product owner from concerns that every last detail must be written on the card. Indeed, this is one of the reasons for using such a lightweight, apparently unimportant medium as index cards. They serve as a constant reminder that the card does not need to hold all the details. The details will come out during conversations between the product owner and the team.

The product owner's reciprocal promise of availability is important because it allows the team to accept work into a sprint without having considered all details, because doing so is impossible anyway. The product owner does not need to be constantly available to the team, although this is helpful and does lead to higher productivity. Rather, what the product owner is promising is to be accessible; it won't take two weeks to schedule a phone call, for example.

"I can't possibly put my requirements on index cards."

That's fine. Projects with distributed teams, very large teams, traceability requirements, or similar needs often require the use of a software tool. A good tool can improve high-level product planning, what-if scenario discussions, and broad communication. However, teams using software tools rather than pen-and-paper are much more likely to struggle with the shift from documents to discussion that Scrum requires. A team using a tool is much more likely to fall into a number of dangerous traps, including

- Writing overly long feature descriptions
- Having only a subset of the team (business analysts) working to understand users' needs
- Resisting the need to split user stories so that complete stories can be delivered within a sprint
- Holding on to stories that are no longer needed because it's actually easier to keep them than to delete them from the tool

I would never go so far as to say you cannot be agile when using a tool to manage your product backlog. I will, however, say that you can be more agile with pen-and-paper than with a tool. Whenever this low-tech option is possible, use it.

"I'm already good with use cases; do I really need to switch to user stories?"

Use cases are an alternative method for expressing the functionality of a system. If you—*and* the rest of the team including the product owner—are good with use cases, there may be no reason to switch. However, use cases were intended to be much larger than is common for a user story.

In *UML Distilled*, Martin Fowler says that use case originator Ivar Jacobsen expects about 20 use cases for a ten person-year project. That's six person-months per use case. Fowler goes on to say he likes smaller use cases, perhaps having 100 for a ten person-year project. Assuming two-week sprints, a six-person team would take more than two sprints for each Jacobsen-sized use case. The same team could finish just over two Fowler-sized use cases per sprint.

This conflicts with data I've collected from dozens of Scrum teams and hundreds of sprints showing that six-person teams average six to nine user stories per two-week sprint. This indicates that Scrum teams do best with units of work that are smaller than a typical use case. So, although you can have use cases on your product backlog, be aware that you'll probably want to write far smaller ones than were intended by their originator.

"We write back-end software that no users ever see, so user stories don't make sense for us."

The word *user* in *user stories* makes the approach sound more limiting than it is. User stories have been successfully applied in all sorts of domains. A story that reads, "*As the loan authorization system, I want to receive all data as valid, well-formed XML so that I don't have to worry about syntax checking,*" is perfectly valid. Additionally, although I find writing stories in the *As a <type of user>, I want <some goal> so that <some reason>* format to be best, it may not be best for all projects. If that syntax doesn't fit what you're developing, write the backlog in another format. I've had success with Feature-Driven Development's feature syntax of

<action> <result> <object>

Examples using this syntax would include the following:

- Assess the risk of a loan.
- Authorize a cash withdrawal from an account.
- Activate the "service now" light on the dashboard.
- Calculate the frequency of haplotypes.

THINGS TO TRY NOW

- ❑ If it doesn't already exist or isn't in good shape, write your product backlog. Invite all project participants to a meeting and do this collaboratively on index cards. Remind attendees that the text on a story card serves as a promise to have a future discussion; not every detail needs to be included.
- ❑ Print all of the documents that were written on the last project or a typical project. With everyone present, discuss how long each took to write and maintain, whether it was used later, and what would have happened (good or bad) if it had not been written. While doing this, create a pile of documents you agree will be useful on the current project and others you can dispense with.
- ❑ If you are currently using a tool to manage your product and sprint backlog, give it up for at least two sprints. At the end of the planned number of sprints, use the retrospective to discuss how it went. See if you can abandon the tool altogether or reduce your reliance on it toward the use of conversation or paper.
- ❑ In your next sprint retrospective, ask team members to write down the software tool they would most like to stop using. When everyone has finished, share the answers. If one or two tools have been consistently named, discuss the pros and cons of eliminating the tool and then consider dumping it.

Progressively Refine Requirements

When starting a new project, the struggles of the previous project are fresh in our minds. In reflecting on those struggles, a common conclusion is that if we'd only tried harder or done more of something, we might have done better. Although this may sometimes be true, in the case of requirements-gathering it is often not. No matter how long or how hard we work at the start of the project to identify all desired features, we cannot succeed. There are always some things that users and developers cannot be expected to think of until they start to see the system take shape.

Emergent Requirements

These features that we cannot identify in advance are called *emergent requirements*. When someone identifies an emergent requirement, she usually announces it to other team members and users by saying, "Seeing that makes me think of this…" or, "That gives me an idea…" or even occasionally, "Holy crap, we never thought about…" There will always be some things that we think of only after we can see the software. One reason Scrum puts so much emphasis on having working code at the end of each sprint is to create a situation where emergent requirements can be discovered sooner rather than later.

Emergent requirements exist on every nontrivial project, and they can cause problems. For example, emergent requirements make it impossible to perfectly predict schedules. Similarly, an up-front design phase will always be imperfect because it will be impossible for the designers to consider the emergent requirements until they do, in fact, emerge.

When using a sequential development process, project managers handle emergent requirements by adding contingency buffers to the plan and by devoting significant energy to proactive risk management. When an emergent requirement appears, it is viewed as a failure of the plan. In contrast, a Scrum team accepts that requirements will emerge, no matter how carefully team members plan. And rather than view emergent requirements as a failure of the plan, they are viewed as a result of planning either too early or in too much detail.

The first step in dealing with emergent requirements is to acknowledge that we cannot think of everything. After acknowledging that some requirements will emerge as we build the system, it is easier to accept the idea that we don't need (and in fact cannot have) a perfect requirements document up front that specifies all the details of the system to be built. In fact, rather than strive for this degree of completeness, we are better off to specify features with different levels of precision based on when the feature will be worked on.

"I understand that things will change—that requirements can emerge. But I need to specify all requirements at the start of the project because the requirements become part of the contract."

OBJECTION

As much as we'd like to lock down requirements in a contract, we can't. We can pretend requirements are locked down and won't change, but some always do. The best contracts reflect this or at least acknowledge that change will happen. Trond Pedersen describes it this way, "Complaining about requirement change is like complaining about the weather. You can't really change the way the world is, but you can find ways to deal with it. Don't make an offering to Thor [the Viking god of thunder] to make the rain stop; get an umbrella."

The Product Backlog Iceberg

Fortunately, it is easy to write a product backlog that contains features written with different levels of detail. The product backlog items that a team will work on soon must be known in sufficient detail that each can be programmed, tested, and integrated within a single sprint. This leads to the user stories at the top of the product backlog being small but reasonably well understood. User stories that are further down are larger and understood in less detail. These *epic* user stories are left large, often known only in enough detail that each can be estimated approximately and then prioritized. This leads the product backlog to take on the shape of an iceberg, as shown in Figure 13.1.

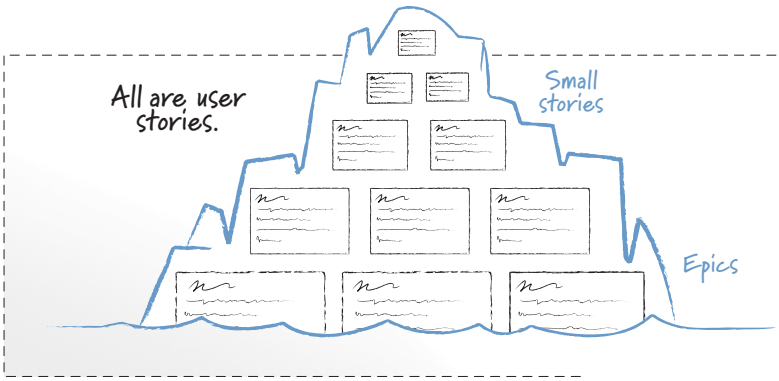

FIGURE 13.1
The product backlog iceberg.

At the top of the product backlog iceberg are the small features the team can fully implement within a sprint. As we look further down the product backlog iceberg (and therefore further into the future), items on the backlog become

increasingly larger until we reach the waterline. The team has no idea what lurks beneath there; those are features that haven't even been discussed yet.

Grooming the Product Backlog

As items are developed and removed from the top of the product backlog, the iceberg develops a flat spot at the top and loses its shape. To counter this effect, time must occasionally be spent *grooming the product backlog*. Grooming the product backlog does not refer to combing its hair. Like me, most product backlogs have no hair. Rather, I use *groom* here in the same sense that this morning's ski report said my local mountains have "groomed, packed powder" and that the *Oxford American Dictionary* defines as meaning "to look after." A team needs to groom, or look after, its product backlog.

A good rule of thumb seems to be that about ten percent of the effort in each sprint should be spent grooming the backlog in preparation for future sprints. This time may come from one individual (perhaps an analyst) whose role on the team is largely focused on the backlog. Or it may represent smaller efforts coming from each team member.

Conversations about the product backlog are not limited to a single time or meeting; they can happen any time and among any team members.

> It's the conversations about user stories that enable developers to understand what needs to be built. [A ScrumMaster needs to] encourage conversations about the user stories to keep happening—before planning meetings, in planning meetings, and after planning meetings. (Davies and Sedley 2009, 75)

Your goal should not be to begin each sprint with a perfect understanding of the product backlog items that will be developed during the sprint. A good Scrum team does not need a perfect understanding of a feature before it starts working on it. Rather, at the start of the sprint, the feature needs only to be sufficiently understood that the team has a reasonably strong chance of finishing it during the sprint. Instead of striving to understand all features up front, we want a just-in-time, just-enough approach to understanding features on the product backlog. Large features are split apart and details added to small features just in time as they move up the backlog. Each is described in just-enough detail that the team can complete it during the sprint.

SEE ALSO

Looking ahead down the product backlog is often done by analysts, user experience designers, and others with similar skills. How to do so is described in Chapter 14, "Sprints," in the sections "Prepare in This Sprint for the Next," and "Work Together Throughout the Sprint."

This is not to say that a team cannot choose to put some time into understanding items further down on the product backlog iceberg. In fact, doing so is often necessary. If the team thinks an item further down the product backlog may have an impact on items above it, it can put some effort into understanding it. This often results in the item being split into multiple, smaller product backlog items. However, given our history of favoring up-front understanding of all features, teams should be careful to make sure there is a real need to better understand an

item before putting more early effort into it than would otherwise be warranted based on the item's position on the product backlog.

"We'll never find time to groom the product backlog. We can barely keep up with our coding tasks."

OBJECTION

Remember that Scrum requires you to plan for change. Time must be budgeted for grooming the product backlog. It may not be needed in every sprint, but you'll need to do it often enough to keep small, sprint-sized items at the top of the product backlog while deferring investment in items that will be worked on further in the future.

Why Progressively Refine Requirements?

It can be comforting to start a new project by identifying "all" of the requirements. However, because every project has some emergent requirements, it can't be done. Fortunately, there are advantages to progressively refining requirements, including the following:

- **Things will change.** Over the course of a project, priorities will shift. Some features that were initially thought to be important will become less so as the system is shown to potential users and customers. Other needs will be discovered and have to be properly prioritized. If we acknowledge that change is inevitable, the advantages of structuring your product backlog like an iceberg become more apparent. The features most likely to change are those that will be done further into the future; to account for the increased likelihood of change, these features are described only at a high level.
- **There's no need.** Novelist E.L. Doctorow has written that "writing a novel is like driving at night in the fog. You can only see as far as your headlights, but you can make the whole trip that way." Software development is the same way. My headlights don't illuminate everything between me and the horizon because they don't need to. They light the way far enough for me to see and respond at the speeds my car can safely travel. The iceberg-shaped product backlog works similarly. Enough visibility is provided into upcoming items that teams see far enough into the future to avoid most issues. The faster a team goes, the further ahead in the product backlog it will need to peer.
- **Time is scarce.** Nearly all projects are time constrained. We want more than will fit in the time allotted. Treating all requirements as equivalent is wasteful. With a limited supply of one of a project's most critical resources (time), we need to be protective of it. If it is sufficient for now to describe

a future feature at a high level, this is all that should be done. When that future feature needs to be better understood—whether because it has moved to the top of the product backlog or because we expect it to influence the implementation of another feature—we can describe it in more detail.

Progressive Refinement of User Stories

An agile requirements process must support the creation of requirements at the various levels shown in the product backlog iceberg of Figure 13.1. Team members must be able to easily create large, placeholder requirements that lie at the bottom of the product backlog iceberg, later disaggregate them into medium-size items, and eventually split them into small-enough pieces that each can be delivered by the team in a single sprint. Just as user stories work well in shifting the emphasis from writing about requirements to talking about them, they also fit well onto the product backlog iceberg. This is because of the ease with which we can move between large and small user stories.

A large user story is typically referred to as an epic. Although there is no magic size at which we start calling a user story an epic, generally an epic is a user story that will take more than one or two sprints to develop and test. Because a team must be able to completely finish a user story within the sprint in which it starts it, this means that epics will be split into smaller user stories before work begins on them. Let's look at an epic and how it may be split into smaller pieces. Consider the following:

- As a user, I am required to log into the system so that my information can only be accessed by me.

This may not appear to be an epic, and it may not be one in all cases. However, for our purposes, let's assume that the product owner clarifies that this simple story is intended to cover everything to do with logging in—requesting a new password, changing the password, and so on. It is about more than pressing a *Login* button on one screen. Based on this, the team decides the story will probably take two or three sprints to develop and test. This makes it an epic. Because it's an epic, it is split into smaller stories, each of which the team thinks can be completed within a single sprint. Here's one possible set of smaller user stories:

- As a registered user, I can log in with my username and password so that I can trust the system.
- As a new user, I want to register by creating a username and password so that the system can remember my personal information.
- As a registered user, I can change my password so that I can keep it secure or make it easier to remember.

- As a registered user, I want the system to warn me if my password is easy to guess so that my account is harder to break into.
- As a forgetful user, I want to be able to request a new password so that I am not permanently locked out if I forget it.
- As a registered user, I do not want to be sure if it was the username, password, or both that was wrong when my login attempt fails so that someone trying to impersonate me will have a harder time doing so.
- As a registered user, I am notified if there have been three consecutive failed attempts to access my account so that I am aware if someone is trying to access my account.

After an epic is split into smaller stories, I recommend that you get rid of it. Delete it from the tool you're using or rip up the index card. You may choose to retain the epic to provide traceability, if that is needed. Or you may choose to retain the epic because it can provide context for the smaller stories created from it. In many cases, the context of the smaller user stories is obvious because the epics should be split in a just-in-time manner as noted earlier. When an epic is ripped up and turned into smaller user stories shortly before the team begins work on it, remembering the context of the small stories is much easier.

Some Epics Are So Large They Split into Epics

In the case of a much larger epic than our password example, the split may occur in multiple steps: first into some medium-sized stories (perhaps epics themselves), then later into smaller ones. As an example of a larger epic, consider this user story from a company developing software for use by large retail stores:

- As a vice president of marketing, I want to review the performance of historical advertising campaigns so that I can identify profitable campaigns worth repeating.

The idea was that the vice president would be able to browse through statistics on various past advertising campaigns and select the best ones to repeat. For example, which worked best: the television ads during *Desperate Housewive*s, the twice-a-day radio ads, the Thursday newspaper inserts, or the e-mail campaign?

It was clear to all involved on this project that this initial story was too large to complete in one of their two-week sprints. So the story was split in two:

- As a vice president of marketing, I want to select the time frame to use when reviewing the performance of past advertising campaigns so that I can identify profitable ones.
- As a vice president of marketing, I want to select which type of campaigns (direct mail, TV, e-mail, radio, and so on) to include when reviewing the performance of historical advertising campaigns.

The team felt that these stories, while smaller, might still be too large to complete within a sprint, so they were split further. The story about selecting the time frame to use was split into three stories:

- As a vice president of marketing, I want to set simple date ranges to be used when reviewing the performance of past advertising campaigns so that I can pick an exact set of dates.
- As a vice president of marketing, I want to select seasons (spring, summer, winter, fall) to be used when reviewing the performance of past advertising campaigns so that I can view trends across multiple years.
- As a vice president of marketing, I want to select a holiday period (Easter, Christmas, and so on) to be used when reviewing the performance of past advertising campaigns so that I can look for trends across multiple years.

After this final split, the team felt the stories were small enough to complete during a sprint and stopped there. Notice though that even these stories may not be trivial to implement. Selecting holiday ranges such as "from Good Friday through Easter Sunday" or from "Thanksgiving until Christmas" will be difficult because the dates move around from year to year. There was a chance the team could have considered these too big.

In many cases it will be possible to go from a large epic near the waterline of the iceberg directly to small, implementation-size stories. Whether you choose to go through the intermediate step of splitting a large epic into multiple smaller epics will be up to you and largely driven by the context of the project.

Adding Conditions of Satisfaction

Eventually stories are small enough that splitting them further is no longer helpful. At this point it is still possible to progressively refine the requirement by adding *conditions of satisfaction* to the user story. A condition of satisfaction is simply a high-level acceptance test that will be true after the user story is complete. As an example, let's reconsider the following story:

- As a vice president of marketing, I want to select a holiday season to be used when reviewing the performance of past advertising campaigns so that I can identify profitable ones.

We've already established that this is small enough for the team to complete in a sprint. So let's continue to progressively refine this requirement by working with the product owner to add its conditions of satisfaction. To do so, we turn the index card over (metaphorically if you're using a product backlog management tool or wiki) and write the following conditions of satisfaction:

- Make sure it works with major retail holidays: Christmas, Easter, President's Day, Mother's Day, Father's Day, Labor Day, New Year's Day.
- Support holidays that span two calendar years (none span three).
- Holiday seasons can be set from one holiday to the next (such as Thanksgiving to Christmas).
- Holiday seasons can be set to be a number of days prior to the holiday.

Progressive refinement by adding conditions of satisfaction helps the team members by telling them the product owner's expectations for that feature. These can be expectations about what will be included and about what will not be. For example, given the conditions of satisfaction for this story, it's clear that we do not need to support Chinese New Year. Although I seize every opportunity to enjoy a spicy Chinese meal, it is not exactly a big shopping holiday here in the United States. Of course, the product owner could have made this even more obvious by explicitly stating, "Does not need to support Chinese New Year." But even that is probably not necessary because the conversations that support this written part of the user story should bring out details such as that.

THINGS TO TRY NOW

- ❑ Convert your existing product backlog to user stories. Print each current product backlog item on an index card. Group similar cards on a large table or flat surface. For high-priority groups of cards, write individual user stories. Be aware that there will probably not be a one-to-one correlation between old product backlog items and new user stories. For lower-priority groups, replace each group with a single epic. Paper clip or staple the old cards behind the new epic, so you'll have them for reference when it's time to split the epic into sprint-size user stories.
- ❑ During your next sprint planning meeting, make sure that each user story you are bringing into the sprint has clearly identified conditions of satisfaction by the time that meeting ends. During the following sprint retrospective, discuss whether having these identified was helpful.

Learn to Start Without a Specification

Because a Scrum team shifts its focus from writing requirements to talking about them and then progressively refines those requirements over the course of the project, the team is left without the comfort of starting with a traditional specification document. Many groups—quality assurance and technical writing foremost among them—will find this very disconcerting. Part of transitioning to Scrum and achieving long-term success with it will be learning how to comfortably get started on a project without a "complete" specification document.

First, I should be clear that the goal is not to throw out what may be a useful document. What we want instead is to use a specification document appropriately. Apart from meeting regulatory or compliance needs, the primary appropriate use of a specification document is to convey information that is best done in writing. Complex or detailed calculations such as might be found in scientific and mathematical applications are good examples, but there can be many others.

One of the dangers of specification documents is that they are seldom kept up to date. Before you write a document, ask yourself if you are willing to commit to updating the document. If not, either think twice about writing it or consider putting an expiration date on the document, similar to the "best if used by" date on a milk carton.

Specify by Example

Another thing you may want to do is change how you write your specifications. Consider specifying a product through examples. Examples are a wonderful way to communicate the desired behavior of a system, especially when augmented with conversations and some small amount of explanatory written text. Gojko Adzic, author of the book *Bridging the Communication Gap*, describes the value of using examples to explain behavior.

> Working with real-world examples helps us communicate better because people will be able to relate to them more easily. It is also easier to spot inconsistencies between realistic examples. Developers, business people, and testers all need to participate in the discussion about examples. Developers learn about the domain and get a solid foundation for implementation. Testers obtain the knowledge they need firsthand, and they can influence the development by suggesting important cases for discussion. (2009, 32)

To see how this works, suppose we are building a system for use within our company that will automatically approve or reject requests for time off. The first thing our product owner wants is for the system to automatically approve requests that are for fewer days than the employee has already accrued. She writes a user story to describe this: *"As an employee, I want a request for up to my earned vacation time to be automatically approved so that I don't need to wait for someone to approve it manually."* The product owner, perhaps working with a tester, then elaborates by providing the examples shown in Table 13.1.

days_accrued	days_requested	approved?
6	5	Yes
5	6	No
5	5	Yes

TABLE 13.1 Examples showing that a request for more time off than has been accrued will not be automatically approved.

The rows of Table 13.1 show different test cases. The first two columns show the test data of those test cases. The final column indicates what the result of the test should be. So, the first row describes an employee who has accrued six days of time off and has requested five days. In the final column we can see that this request should be approved.

This is an admittedly simple illustration of specification by example, so let's see what happens when the product owner writes the next user story: *"As an employee who has been here more than a year, I want automatic approval of a time-off request that is up to five days more than I've currently accrued."* Specifying this through examples, the product owner creates Table 13.2.

days_accrued	days_requested	employed_over_1_year	approved?
10	11	No	No
10	11	Yes	Yes
10	11	No	No
10	15	Yes	Yes
10	16	Yes	No

TABLE 13.2 A slightly more involved example begins to show the power of specification by example.

Table 13.2 is still fairly simple but hopefully it starts to show the power of specification by example.[1] I won't create more detailed examples, but notice how specification by example becomes even more helpful as the scenarios become more complex. For example, in the preceding user stories the number of days accrued was fixed. In many companies time off is accrued monthly. So a request that might be rejected today could be one that would be approved if the date of the desired time off is three months into the future. To specify a situation such as that with examples, we would add to the table such columns as the request date,

1 This example has been intentionally kept simple so as to show how specification by example works. A more thorough implementation of the same example but showing better ways to construct equivalent tables has been provided by Jeff Langr, author of *Agile Java*. His implementation is available at www.informit.com/articles/article.aspx?p=1393274.

the time off start date, and the rate of accrual. Explaining a detailed requirement such as this through a combination of conversations and examples increases the likelihood that what the product owner thinks she's asked for is what the developers build.

Specification by example becomes extremely powerful when the examples can be turned into automated tests. This is not as far-fetched as it may seem. One of the biggest benefits is that we can instantly tell if the specification is out of date—run the automated tests, if they pass then the application conforms to the specification. The tests become self-verifying specifications. They both express detailed design decisions and automatically verify that the application conforms to that specification.

SEE ALSO

FitNesse, available from www.fitnesse.org, is my favorite tool for doing this. It allows you to create and run tests that specify functionality by example almost exactly as shown here. Another popular tool is Cucumber.

This is exactly the approach taken by Trond Wingård, an agile project manager in Norway. Wingård's team made extensive use of FitNesse, a wiki for creating executable tests and specifications in the form of tests. His project's approach is shown in Figure 13.2 and described by Wingård as follows:

> We have a policy that all requirements and tests should be in FitNesse—no exceptions. Even if we find that we need some manual tests, they too should be described in the FitNesse wiki, not somewhere else. The front page contains a list of nine "user epics" linked to a page for each epic. On each of those pages, the epic is described in user story format, followed by a list of each of the user stories that make up this epic. Each story corresponds to an item on the Product Backlog and is linked to the story's page. On a story page, the story is described followed by a grouped list of Conditions of Satisfaction. These COS are grouped, and each group has its own page with FitNesse tests for them. This structure was very easy to set up and easy to grasp and is a real help for the team.

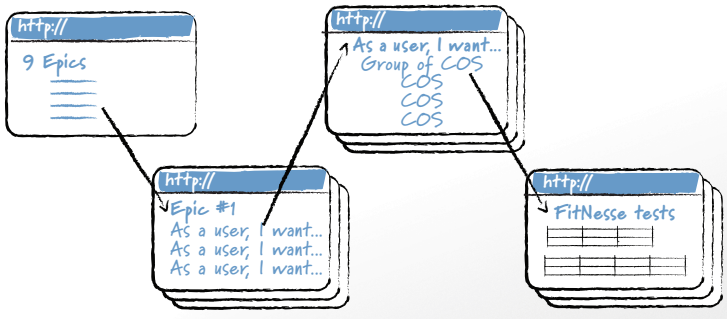

FIGURE 13.2

A series of FitNesse wiki pages that go from high-level requirements written as epics all the way to test cases for each user story.

Cross-Functional Teams Reduce Documentation Needs

A common objection to getting rid of or reducing the scope of specification documents is that these documents are the only way some groups learn what is expected of the system. A QA group, for example, may reason that without a specification document it will not know which behavior is expected and which is buggy. In an organization's pre-Scrum days this would likely have been true. The programmers may have met on their own, made decisions, and then relayed the decisions to the testers through specification documents. After years of exposure to working this way, it would be fair for testers to assert that without the specifications they won't know what to test.

On a Scrum project, however, the programmers and testers work as one team. There is no programming team that hands off work to a testing team. Instead, there is a cross-functional, multidisciplinary team. The testers don't need the same type of documents because work isn't handed off to them in the same way it was in the past. In fact, work isn't handed off to them at all. A tester should be part of the discussion whenever what would have gone in the document is discussed.

Back in my pre-agile days, I often found myself in the middle of arguments between the programmers and testers on a project. The testers complained that programmers weren't keeping documents up to date; the programmers complained that they didn't benefit from the documents. After hearing these same arguments repeated on a handful of projects, I came to the realization that those who benefit from a document should be the ones to write it. Because the testers were the ones who claimed to benefit from the detailed specifications that the programmers were not maintaining, they became the ones responsible for writing and maintaining the document. Not only did this solve my problem, it introduced the additional benefit of forcing the programmers and testers to talk earlier and more frequently so the testers would have the information they needed to write their document. Johannes Brodwall reports using a similar strategy.

> Testers were used to getting very vague, yet very detailed documents and had to try to reinterpret these as test cases. With a more agile approach, the tester is actually the one who's responsible for writing the detailed specification in the first place. Today, we have the tester write the "specification" in terms of testable scenarios at the beginning of an iteration.

Make the Product Backlog DEEP

Roman Pichler, author of *Agile Product Management with Scrum: Creating Products That Customers Love*, and I use the acronym DEEP to summarize key attributes of a good product backlog.

- **Detailed Appropriately.** User stories on the product backlog that will be done soon need to be sufficiently well understood that they can be completed in the coming sprint. Stories that will not be developed for awhile should be described with less detail.
- **Estimated.** The product backlog is more than a list of all work to be done; it is also a useful planning tool. Because items further down the backlog are not as well understood (yet), the estimates associated with them will be less precise than estimates given items at the top.
- **Emergent.** A product backlog is not static. It will change over time. As more is learned, user stories on the product backlog will be added, removed, or reprioritized.
- **Prioritized.** The product backlog should be sorted with the most valuable items at the top and the least valuable at the bottom. By always working in priority order, the team is able to maximize the value of the product or system being developed.

Don't Forget to Talk

Although a project's product backlog will be written somewhere—typically on index cards or entered into a software tool—the product backlog is not a one-to-one replacement for a traditional project's requirements document or use case model. Just as important as what is written in the actual product backlog are the conversations that surround it. These conversations occur when the team and product owner work together to brainstorm items for the initial product backlog And they happen during a sprint as the team and product owner progressively refine their understanding of a feature. In looking to improve your team's use of the product backlog, don't forget the importance of these conversations.

Additional Reading

Adzic, Gojko. 2009. *Bridging the communication gap: Specification by example and agile acceptance testing.* Neuri Limited.

This excellent book describes the reasons why communicating about requirements is difficult. It then proposes specification by example as the solution. Particularly valuable is the chapter on selecting examples. The book also includes a chapter on tools that facilitate specification by example.

Cao, Lan, and Balasubramaniam Ramesh. 2008. Agile requirements engineering practices: An empirical study. *IEEE Software*, January/February, 60–67.

The authors of this academic research paper studied requirements gathering at 16 software development organizations that were using agile approaches. From this study