Chapter 5
Requirements and User Stories

In this chapter I discuss how requirements on a Scrum project are handled differently than on a traditional project. With this context in place, I describe the role of user stories as a common format for representing items of business value. I focus on what user stories are, how they can represent business value at multiple levels of abstraction, and how to determine when the user stories are good. I then describe how to handle nonfunctional requirements and knowledge-acquisition work on a Scrum project. I end by detailing two techniques for gathering user stories.

Overview

Scrum and sequential product development treat requirements very differently. With sequential product development, requirements are nonnegotiable, detailed up front, and meant to stand alone. In Scrum, the details of a requirement are negotiated through conversations that happen continuously during development and are fleshed out *just in time* and *just enough* for the teams to start building functionality to support that requirement.

With sequential product development, requirements are treated much as they are in manufacturing: They are required, nonnegotiable specifications to which the product must conform. These requirements are created up front and given to the development group in the form of a highly detailed document. It is the job of the development group, then, to produce a product that conforms to the detailed requirements.

When a change from the original plan is deemed necessary, it is managed through a formal change control process. Because conformance to specifications is the goal, these deviations are undesirable and expensive. After all, much of the work in process (WIP), in the form of highly detailed requirements (and all work based on them), might need to be changed or discarded.

In contrast, Scrum views requirements as an important degree of freedom that we can manipulate to meet our business goals. For example, if we're running out of time or money, we can drop low-value requirements. If, during development, new information indicates that the cost/benefit ratio of a requirement has become significantly less favorable, we can choose to drop the requirement from the product. And if a new high-value requirement emerges, we have the ability to add it to the product, perhaps discarding a lower-value requirement to make room.

We have probably all had the experience of writing a "complete" requirements document at the beginning of development, only to discover later that an important

requirement was missing. When we discovered that missing requirement, the conversation probably sounded something like this:

> Customer: "Now that I see these built features, I realize I need this other feature that isn't in the requirements document."
>
> Developers: "If you wanted that feature, why didn't you specify it up front?"
>
> Customer: "Well, I didn't realize I needed that feature until I saw the product coming together."
>
> Developers: "Well, if you had thought longer and harder about the requirements up front, you would have found that feature then instead of now."

The fact is, when developing innovative products, you can't create complete requirements or designs up front by simply working longer and harder. Some requirements and design will always emerge once product development is under way; no amount of comprehensive up-front work will prevent that.

Thus, when using Scrum, we don't invest a great deal of time and money in fleshing out the details of a requirement up front. Because we expect the specifics to change as time passes and as we learn more about what we are building, we avoid overinvesting in requirements that we might later discard. Instead of compiling a large inventory of detailed requirements up front, we create placeholders for the requirements, called **product backlog items** (**PBIs**). Each product backlog item represents desirable business value (see Figure 5.1).

Initially the product backlog items are large (representing large swaths of business value), and there is very little detail associated with them. Over time, we flow these product backlog items through a series of conversations among the stakeholders, product owner, and development team, refining them into a collection of smaller, more detailed PBIs. Eventually a product backlog item is small and detailed enough to move into a sprint, where it will be designed, built, and tested. Even during the sprint, however, more details will be exposed in conversations between the product owner and the development team.

As I will discuss in Chapter 6, the product backlog is simply a snapshot of the current collection of product backlog items and their associated details.

While Scrum doesn't specify any standard format for these product backlog items, many teams represent PBIs as user stories. You don't have to. Some teams prefer use cases, and others choose to represent their PBIs in their own custom formats.

In this book, I employ user stories as the principal representation of product backlog items. I will discuss the details of user stories later in this chapter. Even if you choose to use something else, you'll still find the discussion of user stories helpful in understanding what characteristics you'll want from any other representation.

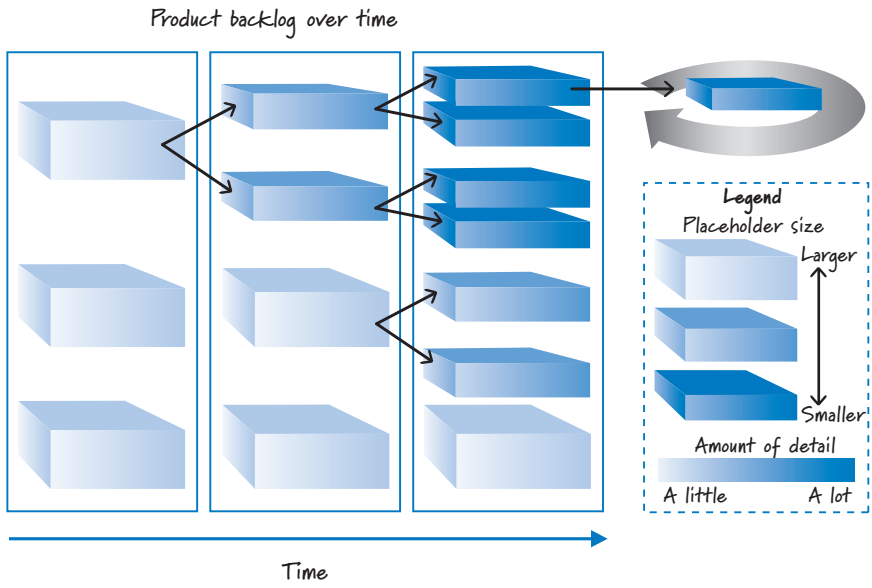

FIGURE 5.1 Scrum uses placeholders for requirements.

Using Conversations

As a communication vehicle, requirements facilitate a shared understanding of what needs to be built. They allow the people who understand what should be created to clearly communicate their desires to the people who have to create it.

Sequential product development relies heavily on written requirements, which look impressive but can easily be misunderstood. I recall a conversation with a VP of Product Management at a company I visited. I asked this person, who managed all of the company's business analysts, how they handled requirements. He said by way of illustration, "On January 1 my team provides the engineering organization with the requirements document, and on December 31 we show up and see what we got."

I asked him who from his team would be available during the year to answer questions and clarify requirements for the developers. He said, "No one. All of the time my group had to invest in this project was spent writing the requirements document. My analysts are off working on the requirements documents for other projects. But don't worry, we wrote a good document, and any questions the developers or testers have can be answered by carefully reading the document."

It seemed unlikely to me that there would be no ambiguities in his 150-page, detailed use case document for a new electronic medical records system. English just isn't that precise; even if it were, people just aren't that precise with their writing.

A way to better ensure that the desired features are being built is for the people who know what they want to have timely conversations with the people who are designing, building, and testing those features.

In Scrum, we leverage conversation as a key tool for ensuring that requirements are properly discussed and communicated. Verbal communication has the benefit of being high-bandwidth and providing fast feedback, making it easier and cheaper to gain a shared understanding. In addition, conversations enable bidirectional communication that can spark ideas about problems and opportunities—discussions that would not likely arise from reading a document.

Conversation, however, is just a tool. It doesn't replace all documents. In Scrum, the product backlog is a "living document," available at all times during product development. Those who still want or must have a requirements specification document can create one at any time, simply by collecting the product backlog items and all of their associated details into a document formatted however they like.

Progressive Refinement

With sequential product development all requirements must be at the same level of detail at the same time. In particular, the approved requirements document must specify each and every requirement so that the teams doing the design, build, and test work can understand how to conform to the specifications. There are no details left to be added.

Forcing all requirements to the same level of detail at the same time has many disadvantages:

- We must predict all of these details early during product development when we have the least knowledge that we'll ever have.
- We treat all requirements the same regardless of their priority, forcing us to dedicate valuable resources today to create details for requirements that we may never build.
- We create a large inventory of requirements that will likely be very expensive to rework or discard when things change.
- We reduce the likelihood of using conversations to elaborate on and clarify requirements because the requirements are already "complete."

As Figure 5.1 illustrates, when using Scrum, not all requirements have to be at the same level of detail at the same time. Requirements that we'll work on sooner will be smaller and more detailed than ones that we won't work on for some time. We employ a strategy of **progressive refinement** to disaggregate, in a just-in-time fashion, large, lightly detailed requirements into a set of smaller, more detailed items.

What Are User Stories?

User stories are a convenient format for expressing the desired business value for many types of product backlog items, especially features. User stories are crafted in a way that makes them understandable to both business people and technical people. They are structurally simple and provide a great placeholder for a conversation. Additionally, they can be written at various levels of granularity and are easy to progressively refine.

As well adapted to our needs as user stories might be, I don't consider them to be the only way to represent product backlog items. They are simply a lightweight approach that dovetails nicely with core agile principles and our need for an efficient and effective placeholder. I use them as the central placeholder to which I will attach any other information that I think is relevant and helpful for detailing a requirement. If I find that user stories are a forced fit for a particular situation (such as representing certain defects), I'll use another approach. For example, I once saw a team write the following user story: "As a customer I would like the system to not corrupt the database." I think we can all agree that a user story is probably not the best way to represent this issue. Perhaps a simple reference to the defect in the defect-tracking system would be more appropriate.

So what exactly are user stories? Ron Jeffries offers a simple yet effective way to think about user stories (Jeffries 2001). He describes them as the three Cs: card, conversation, and confirmation.

Card

The card idea is pretty simple. People originally wrote (and many still do) user stories directly on 3 × 5-inch index cards or sticky notes (see Figure 5.2).

A common template format for writing user stories (as shown on the left in Figure 5.2) is to specify a class of users (the user role), what that class of users wants to achieve (the goal), and why the users want to achieve the goal (the benefit) (Cohn 2004). The "so that" part of a user story is optional, but unless the purpose of the

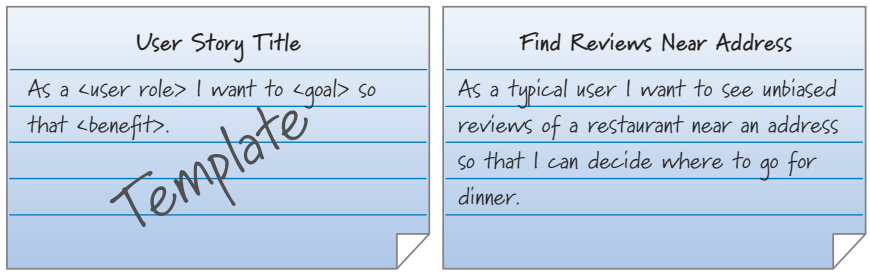

FIGURE 5.2 A user story template and card

story is completely obvious to everyone, we should include it with every user story. The right side of Figure 5.2 shows an example of a user story based on this template.

The card isn't intended to capture all of the information that makes up the requirement. In fact, we deliberately use small cards with limited space to promote brevity. A card should hold a few sentences that capture the essence or intent of a requirement. It serves as the placeholder for more detailed discussions that will take place among the stakeholders, product owner, and development team.

Conversation

The details of a requirement are exposed and communicated in a conversation among the development team, product owner, and stakeholders. The user story is simply a promise to have that conversation.

I say "that conversation," but in actuality, the conversation is typically not a one-time event, but rather an ongoing dialogue. There can be an initial conversation when the user story is written, another conversation when it's refined, yet another when it's estimated, another during sprint planning (when the team is diving into the task-level details), and finally, ongoing conversations while the user story is being designed, built, and tested during the sprint.

One of the benefits of user stories is that they shift some of the focus away from writing and onto conversations. These conversations enable a richer form of exchanging information and collaborating to ensure that the correct requirements are expressed and understood by everyone.

Although conversations are largely verbal, they can be and frequently are supplemented with documents. Conversations may lead to a UI sketch, or an elaboration of business rules that gets written down. For example, I visited an organization that was developing medical imaging software. One of its stories is shown in Figure 5.3.

Notice that the user story references an entire article for future reading and conversation.

So we're not tossing out all of our documents in favor of user stories and their associated story cards. User stories are simply a good starting point for eliciting the initial essence of what is desired, and for providing a reminder to discuss

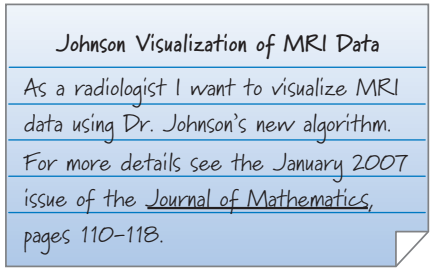

FIGURE 5.3 User story with additional data attached

requirements in more detail when appropriate. However, user stories can and should be supplemented with whatever other written information helps provide clarity regarding what is desired.

Confirmation

A user story also contains confirmation information in the form of conditions of satisfaction. These are acceptance criteria that clarify the desired behavior. They are used by the development team to better understand what to build and test and by the product owner to confirm that the user story has been implemented to his satisfaction.

If the front of the card has a few-line description of the story, the back of the card could specify the conditions of satisfaction (see Figure 5.4).

These conditions of satisfaction can be expressed as high-level acceptance tests. However, these tests would not be the only tests that are run when the story is being developed. In fact, for the handful of acceptance tests that are associated with a user story, the team will have many more tests (perhaps 10 to 100 times more) at a detailed technical level that the product owner doesn't even know about.

The acceptance tests associated with the story exist for several reasons. First, they are an important way to capture and communicate, from the product owner's perspective, how to determine if the story has been implemented correctly.

These tests can also be a helpful way to create initial stories and refine them as more details become known. This approach is sometimes called **specification by example** or **acceptance-test-driven development** (**ATTD**). The idea is fairly intuitive. Discussions about the stories can and frequently do focus on defining specific examples or desired behaviors. For example, in the "Upload File" story in Figure 5.4, the conversation likely went something like this:

> Initially, let's limit uploaded file sizes to be 1 GB or less. Also, make sure that we can properly load common text and graphics files. And for legal reasons we can't have any files with digital rights management (DRM) restrictions loaded to the wiki.

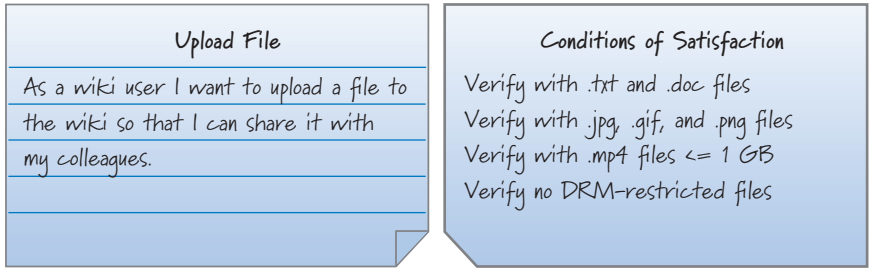

FIGURE 5.4 User story conditions of satisfaction

TABLE 5.1 Automated Test Example

Size	Valid()
0	True
1,073,741,824	True
1,073,741,825	False

If we were using a tool like Fit or FitNesse, we could conveniently define these tests in a table like Table 5.1, which shows examples of different file sizes and whether or not they are valid.

By elaborating on specific examples like these, we can drive the story creation and refinement process and have (automated) acceptance tests available for each story.

Level of Detail

User stories are an excellent vehicle for carrying items of customer or user value through the Scrum value-creation flow. However, if we have only one story size (the size that would comfortably fit within a short-duration sprint), it will be difficult to do higher-level planning and to reap the benefits of progressive refinement.

Small stories used at the sprint level are too small and too numerous to support higher-level product and release planning. At these levels we need fewer, less detailed, more abstract items. Otherwise, we'll be mired in a swamp of mostly irrelevant detail. Imagine having 500 very small stories and being asked to provide an executive-level description of the proposed product to secure your funding. Or try to prioritize among those 500 really small items to define the next release.

Also, if there is only one (small) size of story, we will be obligated to define all requirements at a very fine-grained level of detail long before we should. Having only small stories precludes the benefit of progressively refining requirements on a just-enough, just-in-time basis.

Fortunately, user stories can be written to capture customer and user needs at various levels of abstraction (see Figure 5.5).

Figure 5.5 depicts stories at multiple levels of abstraction. The largest would be stories that are a few to many months in size and might span an entire release or multiple releases. Many people refer to these as **epics**, alluding to the idea that they are *Lord of the Rings* or *War and Peace* size stories. Epics are helpful because they give a very big-picture, high-level overview of what is desired (see Figure 5.6).

We would never move an epic into a sprint for development because it is way too big and not very detailed. Instead, epics are excellent placeholders for a large

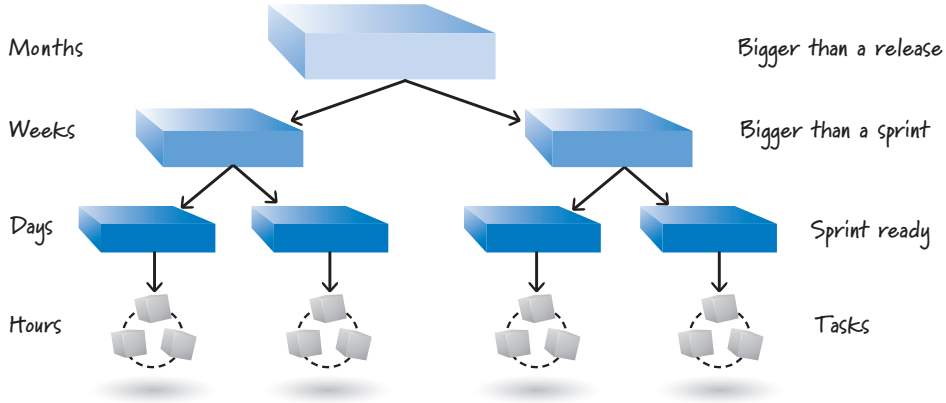

FIGURE 5.5 User story abstraction hierarchy

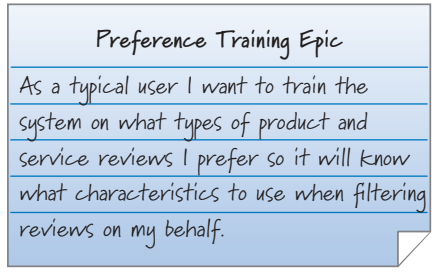

FIGURE 5.6 Example epic

collection of more detailed stories to be created at an appropriate future time. I will illustrate the use of epics during the discussion of product planning in Chapter 17.

The next-size stories in Figure 5.5 are those that are often on the order of weeks in size and therefore too big for a single sprint. Some teams might call these **features**.

The smallest forms of user stories are those I typically refer to as **stories**. To avoid any confusion with epics, features, or other larger items, which are also "stories," some people call these stories either **sprintable stories** or **implementable stories** to indicate that they are on the order of days in size and therefore small enough to fit into a sprint and be implemented. Figure 5.2 provides an example of a sprintable story.

Some teams also use the term **theme** to refer to a collection of related stories. Themes provide a convenient way to say that a bunch of stories have something in common, such as being in the same functional area. In Figure 5.7, the theme represents the collection of stories that will provide the details of how to perform keyword training.

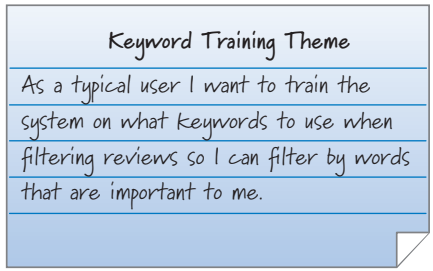

FIGURE 5.7 Example theme

I often think of a theme as the summary card for a bunch of note cards stacked together with a rubber band around them to indicate that they are similar to one another in an area that we think is important.

Tasks are the layer below stories, typically worked on by only one person, or perhaps a pair of people. Tasks typically require hours to perform. When we go to the task layer, we are specifying *how* to build something instead of *what* to build (represented by epics, features, and stories). Tasks are not stories, so we should avoid including task-level detail when writing stories.

It is important to keep in mind that terms like *epic*, *feature*, *story*, and *theme* are just labels of convenience, and they are not universally shared. It really doesn't matter what labels you use as long as you use them consistently. What does matter is recognizing that stories can exist at multiple levels of abstraction, and that doing so nicely supports our efforts to plan at multiple levels of abstraction and to progressively refine big items into small items over time.

INVEST in Good Stories

How do we know if the stories that we have written are good stories? Bill Wake has offered six criteria (summarized by the acronym INVEST) that have proved useful when evaluating whether our stories are fit for their intended use or require some additional work (Wake 2003).

The **INVEST** criteria are ***I****ndependent*, ***N****egotiable*, ***V****aluable*, ***E****stimatable*, ***S****mall* (sized appropriately), and ***T****estable*. When we combine the information derived from applying each criterion, we get a clear picture of what, if any, additional changes we might want to make to a story. Let's examine each criterion.

Independent

As much as is practical, user stories should be *independent* or at least only loosely coupled with one another. Stories that exhibit a high degree of interdependence complicate estimating, prioritizing, and planning. For example, on the left side of Figure 5.8, story #10 depends on many other stories.

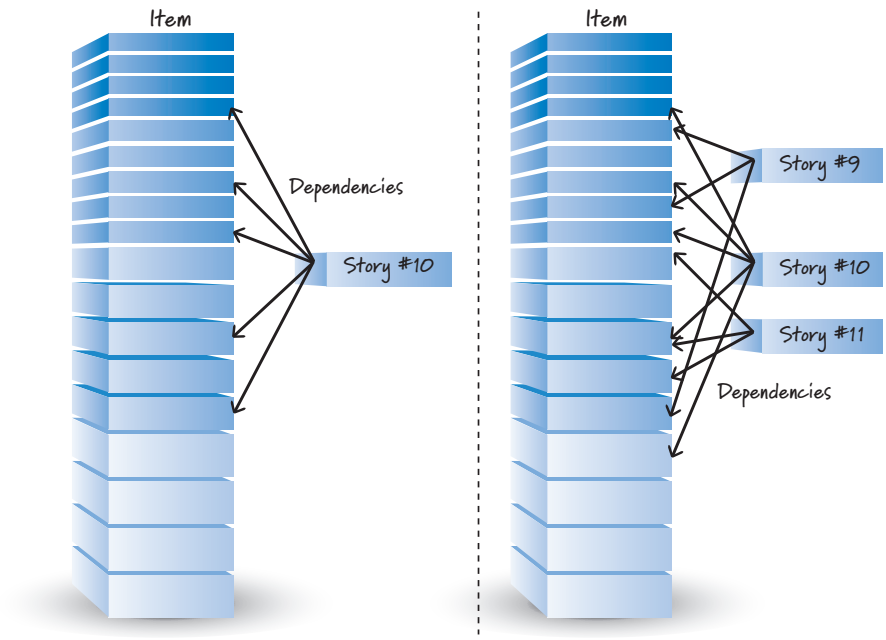

FIGURE 5.8 Highly dependent stories

Before we can work on story #10, we must first develop all of the dependent stories. In this single case that might not be so bad. However, imagine that you have many different stories with a high degree of interdependence, as illustrated by the right side of Figure 5.8. Trying to determine how to prioritize all of these stories and deciding which stories to work on in a sprint would be difficult to say the least.

When applying the *independent* criteria, the goal is not to eliminate all dependencies, but instead to write stories in a way that minimizes dependencies.

Negotiable

The details of stories should also be *negotiable*. Stories are not a written contract in the form of an up-front requirements document. Instead, stories are placeholders for the conversations where the details will be negotiated.

Good stories clearly capture the essence of what business functionality is desired and why it is desired. However, they leave room for the product owner, the stakeholders, and the team to negotiate the details.

This negotiability helps everyone involved avoid the us-versus-them, finger-pointing mentality that is commonplace with detailed up-front requirements documents. When stories are negotiable, developers can't really say, "Hey, if you wanted it, you should have put it in the document," because the details are going to be negotiated with the developers. And the business people can't really say, "Hey, you obviously

didn't understand the requirements document because you built the wrong thing," because the business people will be in frequent dialogue with the developers to make sure there is shared clarity. Writing negotiable stories avoids the problems associated with up-front detailed requirements by making it clear that a dialogue is necessary.

A common example of where negotiability is violated is when the product owner tells the team *how* to implement a story. Stories should be about what and why, not how. When the *how* becomes nonnegotiable, opportunities for the team to be innovative are diminished. The resulting **innovation waste** could have devastating economic consequences.

There are times, however, when *how* something is built is actually important to the product owner. For example, there might be a regulatory obligation to develop a feature in a particular way, or there might be a business constraint directing the use of a specific technology. In such cases the stories will be a bit less negotiable because some aspect of the "how" is required. That's OK; not all stories are fully negotiable, but most stories should be.

Valuable

Stories need to be *valuable* to a customer, user, or both. Customers (or choosers) select and pay for the product. Users actually use the product. If a story isn't valuable to either, it doesn't belong in the product backlog. I can't imagine saying, "Story #10 isn't valuable to anyone, but let's build it anyway." We wouldn't do that. We would either rewrite the story to make it valuable to a customer or user, or we would just discard it.

How about stories that are valuable to the developers but aren't of obvious value to the customers or users? Is it OK to have **technical stories** like the one shown in Figure 5.9?

The fundamental problem with technical stories is that the product owner might not perceive any value in them, making it difficult if not impossible to prioritize them against business-valuable stories. For a technical story to exist, the product owner should understand why he is paying for it and therefore what value it will ultimately deliver.

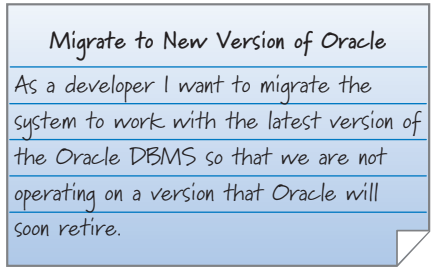

FIGURE 5.9 Example technical story

In the case of the "Migrate to New Version of Oracle" story, the product owner might not initially understand why it is valuable to change databases. However, once the team explains the risks of continuing to develop on an unsupported version of a database, the product owner might decide that migrating databases is valuable enough to defer building some new features until the migration is done. By understanding the value, the product owner can treat the technical story like any other business-valuable story and make informed trade-offs. As a result, this technical story might be included in the product backlog.

In practice, though, most technical stories (like the one in Figure 5.10) should not be included in the product backlog.

Instead, these types of stories should be tasks associated with getting business-valuable stories done. If the development team has a strong definition of done, there should be no need to write stories like these, because the work is implied by the definition of being done.

The crux of the *valuable* criteria is that all stories in the backlog must be valuable (worth investing in) from the product owner's perspective, which represents the customer and user perspectives. Not all stories are independent, and not all stories are fully negotiable, but they all must be valuable.

Estimatable

Stories should be *estimatable* by the team that will design, build, and test them. Estimates provide an indication of the size and therefore the effort and cost of the stories (bigger stories require more effort and therefore cost more money to develop than smaller stories).

Knowing a story's size provides actionable information to the Scrum team. The product owner, for example, needs to know the cost of a story to determine its final priority in the product backlog. The Scrum team, on the other hand, can determine from the size of the story whether additional refinement or disaggregation is required. A large story that we plan to work on soon will need to be broken into a set of smaller stories.

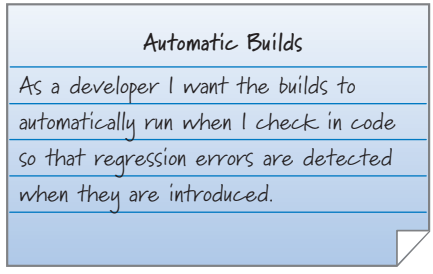

FIGURE 5.10 Undesirable technical story

If the team isn't able to size a story, the story is either just too big or ambiguous to be sized, or the team doesn't have enough knowledge to estimate a size. If it's too big, the team will need to work with the product owner to break it into more manageable stories. If the team lacks knowledge, some form of exploratory activity will be needed to acquire the information (I will discuss this topic shortly).

Sized Appropriately (Small)

Stories should be *sized appropriately* for when we plan to work on them. Stories worked on in sprints should be *small*. If we're doing a several-week sprint, we want to work on several stories that are each a few days in size. If we have a two-week sprint, we don't want a two-week-size story, because the risk of not finishing the story is just too great.

So ultimately we need small stories, but just because a story is large, that doesn't mean it's bad. Let's say we have an epic-size story that we aren't planning to work on for another year. Arguably that story is sized appropriately for when we plan to work on it. In fact, if we spent time today breaking that epic down into a collection of smaller stories, it could easily be a complete waste of our time. Of course, if we have an epic that we want to work on in the next sprint, it's not sized appropriately and we have more work to do to bring it down to size. You must consider *when* the story will be worked on when applying this criterion.

Testable

Stories should be *testable* in a binary way—they either pass or fail their associated tests. Being testable means having good acceptance criteria (related to the conditions of satisfaction) associated with the story, which is the "confirmation" aspect of a user story that I discussed earlier.

Without testable criteria, how would we know if the story is done at the end of the sprint? Also, because these tests frequently provide important story details, they may be needed before the team can even estimate the story.

It may not always be necessary or possible to test a story. For example, epic-size stories probably don't have tests associated with them, nor do they need them (we don't directly build the epics).

Also, on occasion there might be a story that the product owner deems valuable, yet there might not be a practical way to test it. These are more likely to be nonfunctional requirements, such as "As a user I want the system to have 99.999% uptime." Although the acceptance criteria might be clear, there may be no set of tests that can be run when the system is put into production that can prove that this level of uptime has been met, but the requirement is still valuable as it will drive the design.

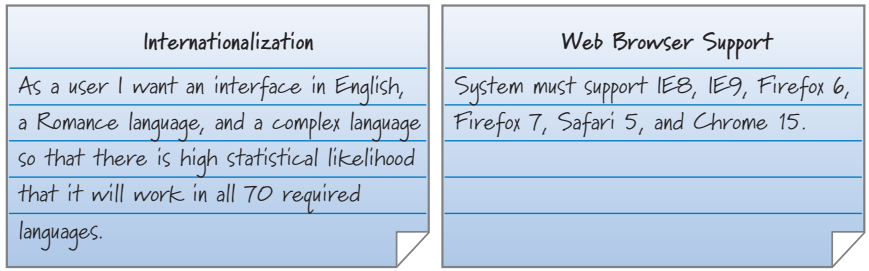

FIGURE 5.11 Nonfunctional requirements

Nonfunctional Requirements

Nonfunctional requirements represent system-level constraints. I frequently write nonfunctional requirements as user stories (see the left side of Figure 5.11), but I feel no obligation to do so, especially if it seems awkward or more convenient to write them in a different format (right side of Figure 5.11).

As system-level constraints, nonfunctional requirements are important because they affect the design and testing of most or all stories in the product backlog. For example, having a "Web Browser Support" nonfunctional requirement (right side of Figure 5.11) would be common on any website project. When the team develops the website features, it must ensure that the site features work with all of the specified browsers.

The team must also decide when to test all of the browsers. Each nonfunctional requirement is a prime target for inclusion in the team's definition of done. If the team includes the "Web Browser Support" nonfunctional requirement in the definition of done, the team will have to test any new features added in the sprint with all of the listed browsers. If it doesn't work with all of them, the story isn't done.

I recommend that teams try to include as many of the nonfunctional requirements in their definitions of done as they possibly can. Waiting to test nonfunctional requirements until late in the development effort defers getting fast feedback on critical system performance characteristics.

Knowledge-Acquisition Stories

Sometimes we need to create a product backlog item that focuses on knowledge acquisition. Perhaps we don't have enough exploitable knowledge about the product or the process of building the product to move forward. So, as I discussed in Chapter 3, we need to explore. Such exploration is known by many names: *prototype*, *proof of concept*, *experiment*, *study*, *spike*, and so on. They are all basically exploration activities that involve buying information.

Often I employ a user story as the placeholder for the exploration work (see Figure 5.12).

In the example, the team wants to evaluate two possible architectures for the new filtering engine. It is proposing to prototype both architectures and then run speed, scale, and type tests against both prototypes. The deliverable from the prototyping activity will be a short memo that describes the experiments that were performed, the results that were obtained, and the team's recommendation for how to proceed.

This specific knowledge-acquisition story looks like a technical story, and as I said earlier, the business value of any technical story has to be justifiable to the product owner. Because product owners think in economic terms, there needs to be an economic justification for doing this prototyping work. There is likely a compelling technical argument for doing a knowledge-acquisition story because the team is typically blocked from making forward progress until it has the knowledge produced by the story. The question for the Scrum team is whether the value of the acquired information exceeds the cost of getting it.

Here is how a Scrum team could approach answering that question. First, we need to know the cost of the prototyping. No good product owner will authorize unbounded exploration. The team might not be able to answer particular questions until an architectural decision has been made, but it must be able to answer the question of how much effort it wants to spend to buy the information necessary to make the architectural decision. So, we ask the team to size the prototyping story.

Let's say that the size estimate indicates that the full team would need to work on the story for one sprint. We know who is on the team and the length of the sprint, so we also know the cost of acquiring the information. (Let's say it is $10K.) Now we need to know the value of the information.

Here is one way we might estimate the value. Imagine that I flip a coin. If it comes up heads, we'll do architecture A; if it comes up tails, we'll do architecture B. Now, I ask the team to estimate the cost of being wrong. For example, if I flip the coin and it comes up heads and we start building business features on top of architecture A, and architecture A turns out to be the wrong approach, what would be the cost to unwind

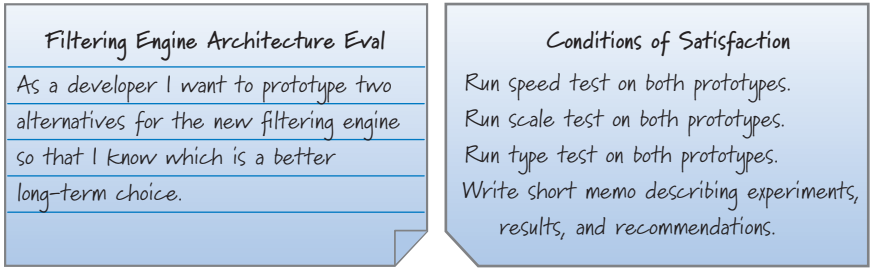

FIGURE 5.12 Knowledge-acquisition story

the bad decision and rebuild everything on top of architecture B? Let's say the team estimates the cost to be $500K.

Now we have enough information to make a sensible economic decision. Are we willing to spend $10K to purchase information that has an expected value of $250K (half the time we flip the coin we would be correct)? Sure, that seems like a sensible business decision. Now the product owner can justify why this story is in the backlog.

As a final illustration of using economics to justify knowledge-acquisition stories, let's alter the numbers. What if the team's response to "What would it cost if we were wrong?" is $15K? In this case it would be a bad decision to do the prototyping story. Why spend $10K to buy information that has an expected value of $7.5K? We would be better off just flipping the coin (or making an educated guess) and, if we're wrong, simply redoing the work using the other architecture. Actually, given today's ever-advancing technologies, this scenario is not as far-fetched as it may sound. It's an example of what some people call a **fail-fast** strategy (try something, get fast feedback, and rapidly inspect and adapt).

Gathering Stories

How do user stories come into existence? Traditional approaches to requirements gathering involve asking the users what they want. I have never been very successful with that approach. In my experience, users are far better critics than they are authors.

So, if you ask a user, "What do you want?" she may or may not be able to answer. Even if she does answer the question and we build exactly what she asked for, she may say, "Yep, you gave me exactly what I asked for, and now that I see it, I want something different." I'm sure we have all had such an experience.

A better approach is to involve the users as part of the team that is determining what to build and is constantly reviewing what is being built. To promote this level of participation, many organizations prefer to employ user-story-writing workshops as a principal means of generating at least the initial set of user stories. Some also employ story mapping to organize and provide a user-centered context to their stories. I will briefly describe each technique.

User-Story-Writing Workshop

The goal of a **user-story-writing workshop** is to collectively brainstorm desired business value and create user story placeholders for what the product or service is supposed to do.

The workshop frequently includes the product owner, ScrumMaster, and development team, in conjunction with internal and external stakeholders. Most workshops last anywhere from a few hours to a few days. I have rarely seen them go longer, nor do I think they should. The goal isn't to generate a full and complete set of user

stories up front (akin to a complete requirements specification on a sequential development project). Instead, the workshop typically has a specific focus. For example, I frequently do a workshop in conjunction with initial release planning to generate a candidate set of stories for the upcoming release (see Chapter 18 for more details).

If it is the first workshop, I usually start by performing user role analysis. The goal is to determine the collection of user roles that can be used to populate the **user role** part of our stories ("As a <user role>, I want to . . ."). Of course, marketing or market research people might have created a good definition of our users in a separate activity prior to the story-writing workshop.

We might also have **personas**, which are prototypical individuals that represent core characteristics of a role. For example, "Lilly," along with her associated description, might be the persona corresponding to the role of the seven- to nine-year-old female player of a young girl's video game. Once Lilly is defined, we would write stories with Lilly in the user role position, instead of a more abstract role such as "Young Female Player." For example, "As Lilly, I want to select from among many different dresses so that I can customize my avatar to my liking."

During the workshop there is no standard way of generating user stories. Some teams prefer to work top-down and others prefer to work bottom-up. The top-down approach involves the team starting with a large story (like an epic) and then concentrating its efforts on generating a reasonable collection of smaller stories associated with the epic.

An alternative is to work more bottom-up and start immediately brainstorming stories that are associated with the next release of an existing system. There isn't a right or wrong approach; use whatever approach works well, or switch approaches to get the best of both.

Story Mapping

Story mapping is a technique popularized by Jeff Patton (Patton 2009) that takes a user-centric perspective for generating a set of user stories. The basic idea is to decompose high-level user activity into a workflow that can be further decomposed into a set of detailed tasks (see Figure 5.13).

Patton uses terms like *activity*, *task*, and *subtask* to describe the hierarchy inside a story map. To be consistent with the terminology I introduced earlier, I use *epic*, *theme*, and *sprintable story*.

At the highest level are the epics, representing the large activities of measurable economic value to the user—for example, the "Buy a Product" epic.

Next we think about the sequence or common workflow of user tasks that make up the epic (represented by themes—collections of related stories). We lay out the themes along a timeline, where themes in the workflow that would naturally occur sooner are positioned to the left of the ones that would occur later. For example, the "Search for Product" theme would be to the left of the "Manage Shopping Cart" theme.

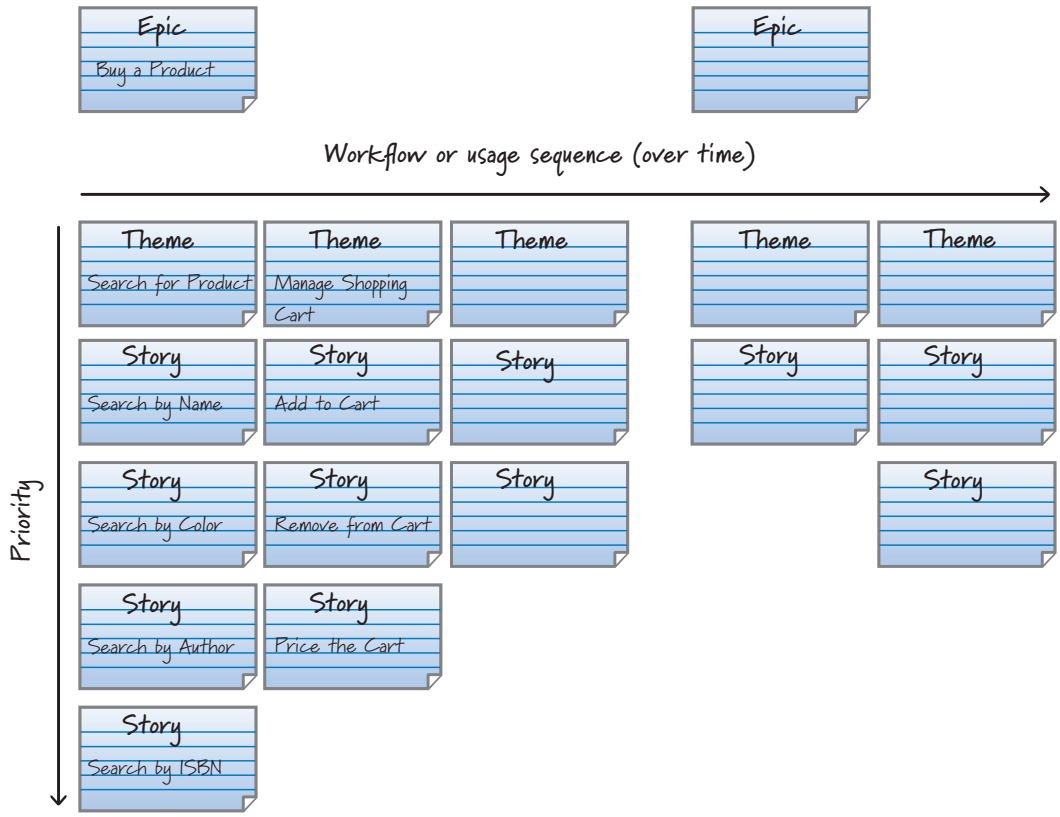

FIGURE 5.13 Story map

Each theme is then decomposed into a set of implementable stories that are arranged vertically in order of priority (really desirability because it is unlikely that the stories have been estimated yet and we can't really know final priority until we know cost). Not all stories within a theme need to be included in the same release. For example, the "Search by Color" story might not be slated for the first release, whereas the "Search by Name" story probably would be.

Story mapping combines the concepts of user-centered design with story decomposition. Good story maps show a flow of activities from the users' perspective and provide a context for understanding individual stories and their relationship to larger units of customer value.

Even if you don't do formal story mapping, I still find the idea of using workflows to be helpful during my story-writing workshops. They focus the discussion on writing stories within the context of delivering a complete workflow of value to the user.

By having the workflow context, it is easier for us to determine if we have missed any important stories associated with the workflow.

One difference between traditional story-writing workshops and story mapping is that during the workshop we are primarily focused on generating stories and not so focused on prioritizing them (the vertical position of implementable stories within a story map). So, we might use story mapping as a complement to the workshop, as a technique for helping to visualize the prioritization of stories. Story maps provide a two-dimensional view of a product backlog instead of the traditional linear (one-dimensional) product backlog representation.

Closing

In this chapter I discussed how requirements are treated differently on a Scrum project than on a traditional, sequential development project. On a development effort that uses Scrum we create placeholders for requirements called product backlog items. These items are frequently expressed as user stories and are flowed through the Scrum process with a distinct focus on conversations as a way of elaborating on the requirements details. We also employ a strategy of progressively refining larger, less detailed stories into smaller, more detailed stories in a just-in-time fashion.

I then formally introduced user stories by describing them in the context of "card, conversation, and confirmation." I went on to discuss how user stories can be used to represent business value at multiple levels of abstraction. Next I explained how the INVEST criteria are helpful in determining whether we have good user stories. Then I introduced ways to deal with nonfunctional requirements and knowledge-acquisition activities. I concluded with a discussion of how to gather user stories, focused on user-story-writing workshops and story mapping. In the next chapter I will discuss the product backlog.