Chapter 6

User Stories

With Pete Behrens

They have been at a great feast of languages, and stol'n the scraps.

—Shakespeare, *Love's Labour's Lost*, Act 5, scene 1

Introduction

In Chapter 3, Agile Requirements for the Team, we introduced the concepts and relationships among the key artifacts—backlogs, user stories, tasks, and so on—used by agile teams to define, build, and test the system of interest. We noted that the user story is the workhorse of agile development, and it is the container that carries the value stream to the user. It also serves as a metaphor for our entire incremental value delivery approach, that is:

> Define a user value story, implement and test it in a short iteration, demonstrate/and or deliver it to the user, repeat forever!

We summarize the requirements artifacts involved in fulfilling this mission in Figure 6–1.

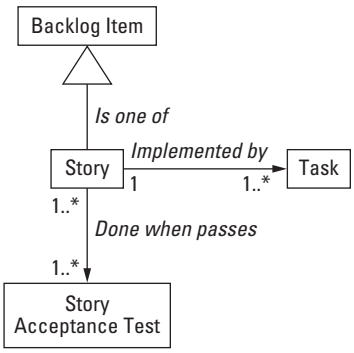


Figure 6–1 Requirements model for teams

From the figure, we see that stories come from the backlog and are implemented via whatever design, coding, and testing tasks are needed to complete the story. Further, stories cannot be considered to be *done* until they pass an associated acceptance test.

In this chapter, we'll describe the user story in more detail, because it is there that we will find the agile practices that help us conform our solution directly to the user's specific needs and help assure quality at the same time.

User Story Overview

We have noted many of the contributions of Scrum to enterprise agile practices, including, for example, the definition of the product owner role, which is integral to our requirements practices. But it is to XP that we owe the invention of the user story, and it is the proponents of XP who have developed the breadth and depth of this artifact. As Beck and Fowler [2005] explain:

> *The story is the unit of functionality in an* XP *project. We demonstrate progress by delivering tested, integrated code that implements a story. A story should be understandable to customers, developer-testable, valuable to the customer, and small enough that the programmers can build half a dozen in an iteration.*

However, though the user story originated in XP, this is less of a "methodological fork in the road" than it might appear, because user stories are now routinely taught within the constructs of Scrum training as a tool for building product backlogs and defining Sprint content. We have Mike Cohn to thank for much of this integration; he has developed user stories extensively in his book *User Stories Applied* [Cohn 2004], and he has been very active in the Scrum community.

For our purposes, we'll define a user story simply as follows:

> A user story is a brief statement of intent that describes something the system needs to do for the user.

In XP, user stories are often written by the customer, thus integrating the customer directly in the development process. In Scrum, the product owner often writes the user stories, with input from the customers, the stakeholders, and the team. However, in actual practice, any team member with sufficient domain knowledge can write user stories, but it is up to the product owner to accept and prioritize these potential stories into the product backlog.

User stories are a tool for defining a system's behavior in a way that is understandable to both the developers and the users. User stories focus the work on the value defined by the user rather than a functional breakdown structure, which is the way work has traditionally been tasked. They provide a lightweight and effective approach to managing requirements for a system.

A user story captures a short statement of function on an index card or perhaps with an online tool. In simple backlog form, stories can just be a list of things the system needs to do for the user. Here's an example:

> Log in to my web energy-monitoring portal.
>
> See my daily energy usage.
>
> Check my current electricity billing rate.

Details of system behavior do not appear in the brief statement; these are left to be developed later through conversations and acceptance criteria between the team and the product owner.

User Stories Help Bridge the Developer–Customer Communication Gap

In agile development, it is the developer's job to speak the language of the user, not the user's job to speak the language of developers. Effective communication is the key, and we need a common language. The user story provides the common language to build understanding between the user and the technical team.

Bill Wake, one of the creators of XP, describes it this way:[1]

> A pidgin language is a simplified language, usually used for trade that allows people who can't communicate in their native language to nonetheless work together. User stories act like this. We don't expect customers or users to view the system the same way that programmers do; stories act as a pidgin language where both sides can agree enough to work together effectively.

With user stories, we don't have to understand each other's language with the degree of proficiency necessary to craft a sonnet; we just need to understand each other enough to know when we have struck a proper bargain!

User Stories Are Not Requirements

Although user stories do most of the work previously done by software requirements specifications, use cases, and the like, they are *materially different* in a number of subtle yet critical ways.

- They are not detailed requirements specifications (something a system shall do) but are rather negotiable expressions of intent (it needs to do something about like this).

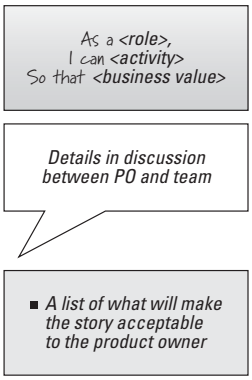
1. *xp123.com/xplor/xp0308/index.shtml*

- They are short, easy to read, and understandable to developers, stakeholders, and users.
- They represent small increments of valued functionality that can be developed in a period of days to weeks.
- They are relatively easy to estimate, so effort to implement the functionality can be rapidly determined.
- They are not carried in large, unwieldy documents but rather organized in lists that can be more easily arranged and rearranged as new information is discovered.
- They are not detailed at the outset of the project but are elaborated on a just-in-time basis, thereby avoiding too-early specificity, delays in development, requirements inventory, and an over-constrained statement of the solution
- They need little or no maintenance and can be safely discarded after implementation.[2,3]
- User stories, and the code that is created quickly thereafter, serve as inputs to documentation, which is then developed incrementally as well.

USER STORY FORM

This section addresses formats for user stories.

Card, Conversation, and Confirmation



Ron Jeffries, another creator of XP, described what has become our favorite way to think about user stories. He used the alliteration *card, conversation, and confirmation* to describe the three elements of a user story.[4]

Card represents two to three sentences used to describe the intent of the story. The card serves as a memorable token, which summarizes intent and represents a more detailed requirement, whose details remain to be determined.

2. This is subject to the development and persistence of acceptance tests, which define the behavior of the system in regression-testable detail.
3. There can be a negative psychological effect to simply discarding the paper cards. One reviewer commented: "Sometimes when we look back at what we'd achieved recently, we don't see visually anything if all the done stories are discarded. So we actually keep all of them, piling them up in the "done" section. It's a bit messy but still it gives a good feeling, we did achieve quite a lot actually…."
4. *xprogramming.com/xpmag/expcardconversationconfirmation/*

▶ **NOTE** In XP and agile, stories are often written manually on physical index cards. More typically in the enterprise, the "card" element is captured as text and attachments in a spreadsheet or agile project management tooling, but teams often still use cards for early planning and brainstorming, as we will see later.

Conversation represents a discussion between the team, customer, product owner, and other stakeholders, which is necessary to determine the more detailed behavior required to implement the intent. In other words, the *card* also represents a "promise for a conversation" about the intent.

Confirmation represents the *acceptance test*, which is how the customer or product owner will confirm that the story has been implemented to their satisfaction. In other words, confirmation represents the *conditions of satisfaction* that will be applied to determine whether the story fulfills the intent as well as the more detailed requirements.

With this simple alliteration, we have an object lesson in how quality in agile is achieved during, rather than after, actual code development. We do that by simply making sure that every new user story is discussed and refined in whatever detail is necessary and is tested to the satisfaction of the key stakeholders.

User Story Voice

In the last few years, a newer, fairly standardized form has been applied that strengthens the user story construct significantly. The form is as follows:

As a <role>, I can <activity> so that <business value>.

where:

- <role> represents who is performing the action or perhaps one who is receiving the value from the activity. It may even be another system, if that is what is initiating the activity.
- <activity> represents the action to be performed by the system.
- <business value> represents the value achieved by the activity.

We call this the *user voice* form of user story expression and find it an exceedingly useful construct[5] because it spans the problem space (<business value> delivered) and the solution space (<activity> the user performs with the system). It also

5. While looking for the origin of this form, I received the following note from Mike Cohn: "It started with a team at Connextra in London and was mentioned at XP2003. I started using it then and wrote about it in my 2004 book, *User Stories Applied*."

provides a user-first (<role>) perspective to the team, which keeps them focused on business value and solving real problems for real people.

This user story form greatly enhances the "why" and "how" understanding that developers need to implement a system that truly meets the needs of the users.

For example, a user of a home energy-management system might want to do the following:

> As a Consumer (<role>), I want to be able to see my daily energy usage (<what I do with the system>) so that I can lower my energy costs and usage (<business value I receive>)."

Each element provides important expansionary context. The *role* allows a segmentation of the product functionality and typically draws out other role-based needs and context for the activity. The *activity* typically represents the "system requirement" needed by the role. And the *value* communicates why the activity is needed, which can often lead the team to finding possible alternative activities that could provide the same value for less effort.

User Story Detail

The details for user stories are conveyed primarily through conversations between the product owner and the team, keeping the team involved from the outset. However, if more details are needed about the story, they can be provided in the form of an attachment (mock-up, spreadsheet, algorithm, or whatever), which is attached to the user story. In that case, the user story serves as the "token" that also carries the more specific behavior to the team. The additional user story detail should be collected over time (just-in-time) through discussions and collaboration with the team and other stakeholders before and during development.

User Story Acceptance Criteria

In addition to the statement of the user story, additional notes, assumptions, and acceptance criteria can be kept with a user story. *Many* discussions about a story between the team and customers will likely take place while the story is being coded. The alternate flows in the activity, acceptance boundaries, and other clarifications should be captured along with the story. Many of these can be turned into acceptance test cases, or other functional test cases, for the story.

Here's an example:

> As a consumer, I want to be able to see my daily energy usage so that I can lower my energy costs and usage.

Acceptance Criteria:

- *Read DecaWatt meter data every 10 seconds and display on portal in 15-minute increments and display on in-home display every read.*
- *Read KiloWatt meters for new data as available and display on the portal every hour and on the in-home display after every read.*
- *No multiday trending for now (another story).*

Etc....

Acceptance criteria are not functional or unit tests; rather, they are the conditions of satisfaction being placed on the system. Functional and unit tests go much deeper in testing all functional flows, exception flows, boundary conditions, and related functionality associated with the story.

INVEST in Good User Stories

Agile teams spend a significant amount of time in discovering, elaborating, and understanding user stories and writing acceptance tests for them. This is as it should be, because it represents the following conclusion:

> Writing the code for an understood objective is not necessarily the hardest part of software development; rather, it is understanding what the real objective for the code *is*.

Therefore, *investing* in good user stories, albeit at the last responsible moment, is a worthy effort for the team. Bill Wake coined the acronym INVEST[6] to describe the attributes of a good user story.

Independent
Negotiable
Valuable
Estimable
Small
Testable

The INVEST model is now fairly ubiquitous, and many agile teams evaluate their stories with respect to these attributes. Here's our view of the value of the team's INVESTment.

6. Bill Wake. *www.XP123.org*.

Independent

Independence means that a story can be developed, tested, and potentially even delivered on its own. Therefore, it can also be independently *valued*.

Many stories will have some natural sequential dependencies as the product functionality builds, and yet each piece can deliver value independently. For example, a product might display a single record and then a list, then sort the list, filter the list, prepare a multipage list, export the list, edit items in the list, and so on. Many of these items have sequential dependencies, yet each item provides independent value, and the product can be potentially shipped through any stopping point of development.

However, many nonvalued dependencies, either technical or functional, also tend to find their way into backlogs, and these we need to find and eliminate. For example, the following might be a nonvalued functional dependency:

> As an administrator, I can set the consumer's password security rules so that users are required to create and retain secure passwords, keeping the system secure.
>
> As a consumer, I am required to follow the password security rules set by the administrator so that I can maintain high security to my account.

In this example, the consumer story depends on the administrator story. The administrator story is testable only in setting, clearing, and preserving the policy, but it is not testable as enforced on the consumer. In addition, completing the administrator story does not leave the product in a potentially shippable state—therefore, it's not independently valuable.

By reconsidering the stories (and the design of the system), we can remove the dependency by splitting the stories in a different manner, in this case through the types of security policies applied and by combining the setup with enforcement in each story:

> As an administrator, I can set the password expiration period so that users are forced to change their passwords periodically.
>
> As an administrator, I can set the password strength characteristics so that users are required to create difficult-to-hack passwords.

Now, each story can stand on its own and can be developed, tested, and delivered independently.

Negotiable . . . and Negotiated

Unlike traditional requirements, a user story is not a contract for specific functionality but rather a placeholder for requirements to be discussed, developed, tested, and accepted. This process of negotiation between the business and the team recognizes the legitimacy and primacy of the business inputs but allows for discovery through collaboration and feedback.

In our prior, siloed organizations, written requirements were generally required to facilitate the limited communication bandwidth between departments and to serve as a record of past agreements. Agile, however, is founded on the concept that a team-based approach is more effective at solving problems in a dynamic collaborative environment. A user story is real-time and structured to leverage this effective and direct communication and collaboration approach.

Finally, the negotiability of user stories helps teams achieve predictability. The lack of overly constraining and too-detailed requirements enhances the team's and business's ability to make trade-offs between functionality and delivery dates. Because each story has flexibility, the team has more flexibility to meet release objectives, which increases dependability and fosters trust.

Valuable

An agile team's goal is simple: to deliver the most value given their existing time and resource constraints. Therefore, value is the most important attribute in the INVEST model, and every user story must provide some value to the user, customer, or stakeholder of the product. Backlogs are prioritized by value, and businesses succeed or fail based on the value the teams can deliver.

A typical challenge facing teams is learning how to write small, incremental user stories that can effectively deliver value. Traditional approaches have taught us to create functional breakdown structures based on technical components. This technical layering approach to building software delays the value delivery until all the layers are brought together after multiple iterations. Wake[7] provides his perspective of vertical, rather than technical, layering:

> Think of a whole story as a multi-layer cake, e.g., a network layer, a persistence layer, a logic layer, and a presentation layer. When we split a story [horizontally], we're serving up only part of that cake. We want to give the customer the essence of the whole cake, and the best way is to slice vertically through the layers. Developers often have an inclination to work on only one layer at a time (and get it "right"); but a full database layer (for example) has little value to the customer if there's no presentation layer.

7. Bill Wake. *www.XP123.org*.

Creating valuable stories requires us to reorient our functional breakdown structures from a horizontal to a vertical approach. We create stories that slice through the architecture so that we can present value to the user and seek their feedback as early and often as possible.

Although normally the value is focused on the user interacting with the system, sometimes the value is more appropriately focused on a customer representative or key stakeholder. For example, perhaps a marketing director is requesting a higher click-through rate on ads presented on the Web site. Although the story could be written from the perspective of the end user…

> As a consumer, I can see other energy pricing programs that appeal to me so that I can enroll in a program that better suits my lifestyle.

…to provide a clearer perspective on the real value, it would be more appropriately written from the marketing director's perspective:

> As a utility marketing director, I can present users with new pricing programs so that they are more likely to continue purchasing energy from me.

Another challenge faced by teams is to articulate value from technical stories such as code refactoring, component upgrades, and so on. For example, how would the product owner determine the value of the following?

> Refactor the error logging system.

Articulating the value of a technical solution as a user story will help communicate to the business its relative importance. Here's an example:

> As a consumer, I can receive a consistent and clear error message anywhere in the product so that I know how to address the issue. OR
>
> As a technical support member, I want the user to receive a consistent and clear message anywhere in the application so they can fix the issue without calling support.

In these latter examples, the value is clear to the user, to the product owner, to the stakeholders, and to the team.

Estimable

A good user story is estimable. Although a story of any size can be in the backlog, in order for it to be developed and tested in an iteration, the team should be able to provide an approximate estimation of its complexity and amount of work required to complete it. The minimal investment in estimation is to determine whether it can be completed within a single iteration. Additional estimation accuracy will increase the team's predictability.

If the team is unable to estimate a user story, it generally indicates that the story is too large or uncertain. If it is *too large* to estimate, it should be split into smaller stories. If the story is *too uncertain* to estimate, then a technical or functional spike story can be used to reduce uncertainty so that one or more estimable user stories result. (Each of these topics is discussed in more detail in the following sections.)

One of the primary benefits of estimating user stories is not simply to derive a precise size but rather to draw out any hidden assumptions and missing acceptance criteria and to clarify the team's shared understanding of the story. Thus, the conversation surrounding the estimation process is as (or more) important than the actual estimate. The ability to estimate a user story is highly influenced by the size of the story, as we'll see shortly.

Small

User stories should be small enough to be able to be completed in an iteration. Otherwise, they can't provide any value or be considered *done* at that point. However, even smaller user stories provide more agility and productivity. There are two primary reasons for this: *increased throughput* and *decreased complexity*.

Increased Throughput

From queuing theory, we know that smaller batch sizes go through a system faster. This is one of the primary principles of lean flow and is captured in Little's law:

$$\text{Cycle Time} = \frac{\text{Work In Process}}{\text{Throughput}}$$

In a stable system (where throughput, the amount of work that can be done in a unit of time, is constant), we have to decrease work in process (the amount of things we are working on) in order to decrease cycle time (the time elapsed between the beginning and end of the process). In our case, that means *fewer, smaller stories in process will come out faster*.

Moreover, when a system is loaded to capacity, it can become unstable, and the problem is compounded. In heavily loaded systems, larger batches move disproportionately slower (throughput decreases) through the system. (Think of a highway system at rush hour. Motorcycles and bicycles have a much higher throughput than do cars and trucks. There is more space to maneuver smaller things through a loaded system.) Because development teams are typically fully allocated at or above capacity (80% to 120%), they fall in the "rush-hour highway" category.

When utilization hits 80% or so, larger objects increase cycle time (slow down) much more than smaller objects. Worse, the *variation* in cycle time increases, meaning

that it becomes harder to predict when a batch might actually exit the system, as shown in Figure 6–2. In turn, this lower predictability wreaks havoc with schedules, commitments, and the credibility of the team.

Decreased Complexity

Smaller stories not only go through faster because of their raw, proportional size, but they go through faster yet because of their *decreased complexity*, and complexity has a *nonlinear relationship* to size. This is seen most readily in testing, where the permutations of tests required to validate the functionality increase at an exponential rate with the complexity of the function itself. This correlates to the advice we receive about developing clean code, as Robert Martin [2009] notes on his rules for writing software functions.

- Rule 1: Do one thing.
- Rule 2: Keep them small.
- Rule 3: Make them smaller than that.

This is one of the primary reasons that the Fibonacci estimating sequence (that is, 1, 2, 3, 5, 8, 13, 21…) is so effective in estimating user stories. The effort estimate grows nonlinearly with increasing story size.

On the Relationship of Size and Independence

A fair question arises as to the relationship between size and independence, because it seems logical that smaller stories increase the number of dependencies. However, smaller stories, even with some increased dependency, deliver higher value through-

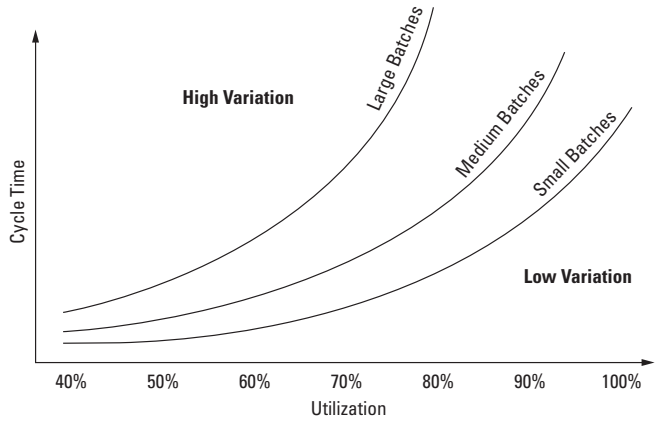


Figure 6–2 Large batches have higher cycle times and higher cycle time variability [Poppendieck and Poppendieck 2007].

put and provide faster user feedback than larger stories. So, the agilist always leans to smaller stories *and then makes them smaller still.*

Testable

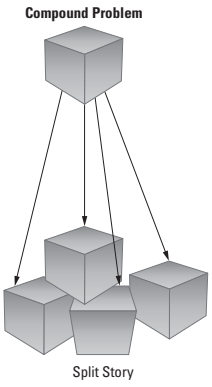
In proper agile, *all code is tested code*, so it follows that stories must be testable. If a story does not appear to be testable, then the story is probably ill-formed, overly complex, or perhaps dependent on other stories in the backlog.

To assure that *stories don't get into an iteration if they can't get out* (be successfully tested), many agile teams today take a "write-the-test-first" approach. This started in the XP community using test-driven development, a practice of writing automated unit tests prior to writing the code to pass the test.

Since then, this philosophy of approach is being applied to development of story acceptance criteria and the necessary functional tests prior to coding the story itself. If a team really knows how to test a story, then they likely know how to code it as well.

To assure testability, user stories share some common testability pitfalls with requirements. Vague words such as *quickly*, *manage*, *nice*, *clean*, and so on, are easy to write but very difficult to test because they mean different things to different people and therefore should be avoided. And although these words do provide negotiability, framing them with some clear boundaries will help the team and the business share expectations of the output and avoid big surprises.

SPLITTING USER STORIES



User stories are often driven by epics and features—a large, vague concept of something we want to do for a user. We often find these big-value stories during our discovery process and capture them in the backlog. However, these are *compound stories*, as pictured on the left, and are usually far too big to be implemented within an iteration. To prepare the work for iterations, a team must break them down into smaller stories.

There is no set routine for splitting user stories into iteration-sized bites, other than the general guidance to make each story provide a vertical slice, some piece of user value, through the system. However, we recommend applying an appropriate selection of *ten common patterns to split a user story*, as Table 6–1 indicates.[8]

8. Adapted from Richard Lawrence, *www.richardlawrence.info/2009/10/28/patterns-for-splitting-user-stories/*

Table 6–1 Ten Patterns for Splitting a User Story

1. Workflow Steps
Identify specific steps that a user takes to accomplish a specific workflow, and then implement the workflow in incremental stages.

As a utility, I want to update and publish pricing programs to my customer.	…I can publish pricing programs to the customer's in-home display.
	…I can send a message to the customer's web portal.
	…I can publish the pricing table to a customer's smart thermostat.

2. Business Rule Variations
At first glance, some stories seem fairly simple. However, sometimes the business rules are more complex or extensive than the first glance revealed. In this case, it might be useful to break the story into several stories to handle the business rule complexity.

As a utility, I can sort customers by different demographics.	…sort by ZIP code.
	…sort by home demographics.
	…sort by energy consumption.

3. Major Effort
Sometimes a story can be split into several parts where most of the effort will go toward implementing the first one. In the example shown next, processing infrastructure should be built to support the first story; adding more functionality should be relatively trivial later.

As a user, I want to be able to select/change my pricing program with my utility through my web portal.	…I want to use time-of-use pricing.
	…I want to prepay for my energy.
	…I want to enroll in critical-peak pricing.

4. Simple/Complex
When the team is discussing a story and the story seems to be getting larger and larger ("What about x? Have you considered y?"), stop and ask, "What's the simplest version that can possibly work?" Capture that simple version as its own story, and then break out all the variations and complexities into their own stories.

As a user, I basically want a fixed price, but I also want to be notified of critical-peak pricing events.	…respond to the time and the duration of the critical-peak pricing event.
	…respond to emergency events.

5. Variations in Data
Data variations and data sources are another source of scope and complexity. Consider adding stories just-in-time after building the simplest version. A localization example is shown here:

As a utility, I can send messages to customers.	…customers who want their messages:
	…in Spanish
	…in Arabic, and so on.

6. Data Entry Methods
Sometimes complexity is in the user interface rather than the functionality itself. In that case, split the story to build it with the simplest possible UI, and then build the richer UI later.

As a user, I can view my energy consumption in various graphs.	…using bar charts that compare weekly consumption. …in a comparison chart, so I can compare my usage to those who have the same or similar household demographics.

7. Defer System Qualities
Sometimes, the initial implementation isn't all that hard, and the major part of the effort is in making it fast or reliable or more precise or more scalable. However, the team can learn a lot from the base implementation, and it should have some value to a user, who wouldn't otherwise be able to do it all. In this case, break the story into successive "ilities."

As a user, I want to see real-time consumption from my meter.	…interpolate data from the last known reading. …display real-time data from the meter.

8. Operations (Example: Create Read Update Delete (CRUD))
Words like *manage* or *control* are a giveaway that the story covers multiple operations, which can offer a natural way to split the story.

As a user, I can manage my account.	…I can sign up for an account. …I can edit my account settings. …I can cancel my account. …I can add more devices to my account.

9. Use-Case Scenarios
If use cases have been developed to represent complex user-to-system or system-to-system interaction, then the story can often be split according to the individual scenarios of the use case.*

I want to enroll in the energy savings program through a retail distributor.	Use case/story #1 (happy path): Notify utility that consumer has equipment. Use case/story #2: Utility provisions equipment and data and notifies consumer. Use case/story #3 (alternate scenario): Handle data validation errors.

10. Break Out a Spike
In some cases, a story may be too large or overly complex, or perhaps the implementation is poorly understood. In that case, build a technical or functional spike to figure it out; then split the stories based on that result. (See the "Spikes" section.)

*The application of use cases in agile development is the entire topic of Chapter 19.

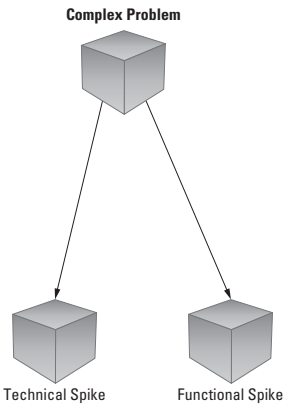
When splitting stories, the team should use an appropriate combination of the previous techniques to consider means of decomposition or multiple patterns in combination. With this skill, the team will be able to move forward at a more rapid pace, splitting user stories at release- and iteration-planning boundaries into bite-size chunks for implementation.

SPIKES

Spikes, another invention of XP, are a special type of story used to drive out risk and uncertainty in a user story or other project facet. Spikes may be used for a number of reasons.

- Spikes may be used for basic research to familiarize the team with a new technology or domain.
- The story may be too big to be estimated appropriately, and the team may use a spike to analyze the implied behavior so they can split the story into estimable pieces.
- The story may contain significant technical risk, and the team may have to do some research or prototyping to gain confidence in a technological approach that will allow them to commit the user story to some future timebox.
- The story may contain significant functional risk, in that although the intent of the story may be understood, it's not clear how the system needs to interact with the user to achieve the benefit implied.

Technical Spikes and Functional Spikes



Technical spikes are used to research various technical approaches in the solution domain. For example, a technical spike may be used to determine a build-versus-buy decision, to evaluate potential performance or load impact of a new user story, to evaluate specific implementation technologies that can be applied to a solution, or for any reason when the team needs to develop a more confident understanding of a desired approach before committing new functionality to a timebox.

Functional spikes are used whenever there is significant uncertainty as to how a user might interact with the system. Functional spikes are often best evaluated through some level of prototyping, whether it be user interface mock-ups, wireframes, page flows, or whatever techniques are best suited to get feedback from the customer or stakeholders. Some user stories may require both types of spikes. Here's an example:

> As a consumer, I want to see my daily energy use in a histogram so that I can quickly understand my past, current, and projected energy consumption.

In this case, a team might create two spikes:

> Technical spike: Research how long it takes to update a customer display to current usage, determining communication requirements, bandwidth, and whether to push or pull the data.

> Functional spike: Prototype a histogram in the web portal and get some user feedback on presentation size, style, and charting attributes.

Guidelines for Spikes

Since spikes do not directly deliver user value, they should be used sparingly and with caution. The following are some guidelines for applying user spikes.

Estimable, Demonstrable, and Acceptable

Like other stories, spikes are put in the backlog, estimated, and sized to fit in an iteration. Spike results are different from a story, because they generally produce information, rather than working code. A spike may result in a decision, prototype, storyboard, proof of concept, or some other partial solution to help drive the final results. In any case, the spike should develop just the information sufficient to resolve the uncertainty in being able to identify and size the stories hidden beneath the spike.

The output of a spike is demonstrable, both to the team and to any other stakeholders. This brings visibility to the research and architectural efforts and also helps build collective ownership and shared responsibility for the key decisions that are being taken.

And, like any other story, spikes are accepted by the product owner when the acceptance criteria for the spike have been fulfilled.

The Exception, Not the Rule

Every user story has uncertainty and risk—this is the nature of agile development. The team discovers the right solution through discussion, collaboration, experimentation, and negotiation. Thus, in one sense, every user story contains spike-level activities to flush out the technical and functional risk. The goal of an agile team is to learn how to embrace and effectively address this uncertainty in each iteration. A spike story, on the other hand, should be reserved for the more critical and larger unknowns.

When considering a spike for future work, first consider ways to split the story through the strategies discussed earlier. Use a spike as a last option.

Implement the Spike in a Separate Iteration from the Resulting Stories

Since a spike represents uncertainty in one or more potential stories, planning for both the spike and the resultant stories in the same iteration is risky and should generally be avoided. However, if the spike is small and straightforward and a quick solution is likely to be found, there is nothing wrong with completing the stories in the same iteration. Just be careful.

STORY MODELING WITH INDEX CARDS

Writing and modeling user stories using physical index cards provides a powerful visual and kinesthetic means for engaging the entire team in backlog development. This interactive approach has a number of advantages.

- The physical size of index cards forces a text length limit, requiring the writer to articulate their ideas in just a sentence or two. This helps keep user stories small and focused, which is a key attribute. Also, the tangible and physical nature of the cards gives teams the ability to visually and spatially arrange them in various configurations to help define the backlog.
- Cards may be arranged by feature (or epic) and may be written on the same colored cards as the feature for visual differentiation.
- Cards can also be arranged by size to help developers "see" the size relationships between different stories.
- Cards can be arranged by time or iteration to help evaluate dependencies, understand logical sequencing, see the impact on team velocity, and better align and communicate differing stakeholder priorities.
- The more cards you have, the more work you see, so scoping is a more natural process.

Any team member can write a story card, and the physical act of moving these small, tangible "value objects" around the table creates an interactive learning setting where participants "see and touch" the value they are about to create for their stakeholders.

Experience has shown that teams with a shared vision are more committed to implementing that vision. Modeling value delivery with physical story cards pro-

vides a natural engagement model for all team members and stakeholders—one that results in a shared, tangible vision for all to see and experience.

SUMMARY

In this chapter, we provided an overview of the derivation and application of user stories as the primary requirements proxy used by agile teams. Along with background and history, we described the alliteration *card, conversation, and confirmation*, which defines the key elements of a user story. We provided some recommendations for developing good user stories in accordance with the INVEST model and specifically described how *small* stories increase throughput and quality. We also described a set of patterns for splitting large stories into smaller stories so that each resultant story can independently deliver value in an iteration. We also provided guidelines for creating spikes as story-like backlog items for understanding and managing development risk. In conclusion, we suggested that teams apply visual modeling using physical index cards for developing user stories and create a shared vision for implementing user value using this uniquely agile requirements construct.

In the next chapter, we'll strive for a deeper understanding of the users and user personas for whom these user stories are intended.

This page intentionally left blank