The Addison Wesley Signature Series

A Kent Beck Signature Book

User Stories Applied

For Agile Software Development

Mike Cohn

Foreword by Kent Beck

Table of Contents

Chapter 13

Why User Stories?

With all of the available methods for considering requirements, why should we choose user stories? This chapter looks at the following advantages of user stories over alternative approaches:

- User stories emphasize verbal communication.
- User stories are comprehensible by everyone.
- User stories are the right size for planning.
- User stories work for iterative development.
- User stories encourage deferring detail.
- User stories support opportunistic design.
- User stories encourage participatory design.
- User stories build up tacit knowledge.

After having considered the advantages of user stories over alternative approaches, the chapter concludes by pointing out a few potential drawbacks to user stories.

Verbal Communication

Humans used to have such a marvelous oral tradition; myths and history were passed orally from one generation to the next. Until an Athenian ruler started writing down Homer's *The Iliad* so that it would not be forgotten, stories like Homer's were told, not read. Our memories must have been a lot better back then and must have started to fade sometime in the 1970s because by then we could no longer remember even short statements like "The system shall prompt the user for a login name and password." So, we started writing them down.

And that's where we started to go wrong. We shifted focus to a shared document and away from a shared understanding.

It seems so easy to think that if everything is written down and agreed to then there can be no disagreements, developers will know exactly what to build, testers will know exactly how to test it, and, most importantly, customers will get exactly what they wanted. Well, no, that's wrong: Customers will get the developers' interpretation of what was *written down*, which may not be exactly what they *wanted*.

Until trying it, it would seem simple enough to write down a bunch of software requirements and have a team of developers build exactly what you want. However, if we have trouble writing a lunch menu with sufficient precision, think how hard it must be to write software requirements. At lunch the other day my menu read:

> Entrée comes with choice of soup or salad and bread.

That should not have been a difficult sentence to understand but it was. Which of these did it mean I could choose?

> Soup or (Salad and Bread)
> (Soup or Salad) and Bread

We often act as though written words are precise, yet they aren't. Contrast the words written on that menu with the waitress' spoken words: "Would you like soup or salad?" Even better, she removed all ambiguity by placing a basket of bread on the table before she took my order.

Just as bad is that words can take on multiple meanings. As an extreme example, consider these two sentences:

> Buffalo buffalo buffalo.
> Buffalo buffalo Buffalo buffalo.

Wow. What can those sentences possibly mean? Buffalo can mean either the large furry animal (also known as a bison), or a city in New York, or it can mean "intimidate," as in "The developers were buffaloed into promising an earlier delivery date." So, the first sentences means that bison intimidate other bison. The second sentence means that bison intimidate bison from the city of Buffalo.

Unless we're writing software for bison this is an admittedly unlikely example; but is it really much worse than this typical requirements statement:

- The system should prominently display a warning message whenever the user enters invalid data.

Does *should* mean the requirement can be ignored if we want? I *should* eat three servings of vegetables a day; I don't. What does *prominently display* mean? What's prominent to whoever wrote this may not be prominent to whoever codes and tests it.

As another example, I recently came across this requirement that was referring to a user's ability to name a folder in a data management system:

- The user can enter a name. It can be 127 characters.

From this statement it is not clear if the user must enter a name for the folder. Perhaps a default name is provided for the folder. The second sentence is almost completely meaningless. Can the folder name be another length or must it always be 127 characters?

Writing things down does have advantages: written words help overcome the limitations of short-term memory, distractions, and interruptions. But, so many sources of confusion—whether from the imprecision of written words or from words with multiple meanings—go away if we shift the focus from writing requirements down to talking about them.

Naturally, some of the problems of our language exist with verbal as well as with written communication; but when customers, developers and users talk there is the opportunity for a short feedback loop that leads to mutual learning and understanding. With conversations there is not the false appearance of precision and accuracy that there is with written words. No one signs off on a conversation and no one points to it and says, "Right there, three months ago on a Tuesday, you said passwords could not contain numbers."

Our goal with user stories is not to document every last detail about a desired feature; rather, it is to write down a few short placeholding sentences that will remind developers and customers to hold future conversations. Many of my conversations occur through email and I couldn't possibly do my job without it. I send and receive hundreds of emails every day. But, when I need to talk to someone about something complicated, I invariably pick up the phone or walk to the person's office or workspace.

A recent conference on traditional requirements engineering included a half-day tutorial on writing "perfect requirements" and promised to teach techniques for writing better sentences to achieve perfect requirements. Writing *perfect* requirements seems like such a lofty and unattainable goal.

And even if each sentence in a requirements document is perfect, there are still two problems. First, users will refine their opinions as they learn more about the software being developed. Second, there is no guarantee that the sum of these perfect parts is a perfect whole. Tom Poppendieck has reminded me that 100 perfect left shoes does not yield a single perfect pair of shoes. A far

more valuable goal than perfect requirements is to augment *adequate stories* with *frequent conversations*.

User Stories Are Comprehensible

One of the advantages that use cases and scenarios bring us over IEEE 830-style software requirements specifications is that they are understandable by both users and developers. IEEE 830-style documents often contain too much technical jargon to be readable by users and too much domain-specific jargon to be readable by developers.

Stories take this further and are even more comprehensible than use cases or scenarios. Constantine and Lockwood (1999) have observed that the emphasis scenarios place on realism and detail can cause scenarios to obscure broader issues. This makes it more difficult when working with scenarios to understand the basic nature of the interactions. Because stories are terse and are always written to show customer or user value, they are always readily comprehensible by business people and developers.

Additionally, a study in the late 1970s found that people are better able to remember events if they are organized into stories (Bower, Black and Turner 1979). Even better, study participants had better recall of both stated actions as well as inferred actions. That is, not only do stories facilitate recall of stated actions, they facilitate recall of the unstated actions. The stories we write can be more terse than traditional requirements specifications or even use cases, and because they are written and discussed as stories, recall will be greater.

User Stories Are the Right Size for Planning

Additionally, user stories are the right size for planning—not too big, not too small, but just right. At some point in the career of most developers it has been necessary to ask a customer or user to prioritize IEEE 830-style requirements. The usual result is something like 90% of the requirements are mandatory, 5% are very desirable but can be deferred briefly, and another 5% may be deferred a bit longer. This is because it's hard to prioritize and work with thousands of sentences all starting with "The system shall…" For example, consider the following sample requirements:

4.6) The system shall allow a room to be reserved with a credit card.
  4.6.1) The system shall accept Visa, MasterCard and American Express cards.
    4.6.1.1) The system shall verify that the card has not expired.
  4.6.2) The system shall charge the credit card the indicated rate for all nights of the stay before the reservation is confirmed.
4.7) The system shall give the user a unique confirmation number.

Each level of nesting within an IEEE 830 requirements specification indicates a relationship between the requirements statements. In the example above, it is unrealistic to think that a customer could prioritize 4.6.1.1 separately from 4.6.1. If items cannot be prioritized or developed separately, perhaps they shouldn't be written as separate items. If they are only written separately so that each may be discretely tested, it would be better to just write the tests directly.

When you consider the thousands or tens of thousands of statements in a software requirements specification (and the relationships between them) for a typical product, it is easy to see the inherent difficulty in prioritizing them.

Use cases and interaction design scenarios suffer from the opposite problem—they're just too big. Prioritizing a few dozen use cases or scenarios is sometimes easy but the results are often not useful because it is rarely the case that all of the top priority items are more important than all of the second priority items. Many projects have tried to correct this by writing many smaller use cases with the result that they swing too far in that direction.

Stories, on the other hand, are of a manageable size such that they may be conveniently used for planning releases, and by developers for programming and testing.

User Stories Work for Iterative Development

User stories also have the tremendous advantage that they are compatible with iterative development. I do not need to write all of my stories before I begin coding the first. I can write some stories, code and test those stories, and then repeat as often as necessary. When writing stories I can write them at whatever level of detail is appropriate. Stories work well for iterative development because of how easy it is to iterate over the stories themselves.

For example, if I'm just starting to think about a project, I may write epic stories like "the user can compose and send email." That may be just right for very early planning. Later I'll split that story into perhaps a dozen other stories:

- A user can compose an email message.
- A user can include graphics in email messages.
- A user can send email messages.
- A user can schedule an email to be sent at a specific time.

Scenarios and IEEE 830 documents do not lend themselves to this type of progressive levels of detail. By the way they are written, IEEE 830 documents imply that if there is no statement saying "The system shall…" then it is assumed that the system shall not. This makes it impossible to know if a requirement is missing or simply has not been written yet.

The power of scenarios is in their detail. So, the idea of starting a scenario without detail and then progressively adding detail as it is needed by the developers makes no sense and would strip scenarios of their usefulness entirely.

Use cases can be written at varying progressive levels of detail, and Cockburn (2001) has suggested excellent ways of doing so. However, rather than writing use cases with free-form text, most organizations define a standard template. The organization then mandates that all use cases conform to the template. This becomes a problem when many people feel compelled to fill in each space on a form. Fowler (1997) refers to this as *completism*. In practice, few organizations are able to write some use cases at a summary level and some at a detailed level. User stories work well for completists because—so far—no one has proposed a template of fields to be written for each story.

Stories Encourage Deferring Detail

Stories also have the advantage that they encourage the team to defer collecting details. An initial place-holding goal-level story ("A Recruiter can post a new job opening") can be written and then replaced with more details once it becomes important to have the details.

This makes user stories perfect for time-constrained projects. A team can very quickly write a few dozen stories to give them an overall feel for the system. They can then plunge into the details on a few of the stories and can be coding much sooner than a team that feels compelled to complete an IEEE 830-style software requirements specification.

Stories Support Opportunistic Development

It is tempting to believe that we can write down all of the requirements for a system and then think our way to a solution in a top-down manner. Nearly two decades ago Parnas and Clements (1986) told us that we will never see a project work this way, because:

- Users and customers do not generally know exactly what they want.
- Even if the software developers know all the requirements, many of the details they need to develop the software become clear only as they develop the system.
- Even if all the details could be known up front, humans are incapable of comprehending that many details.
- Even if we could understand all the details, product and project changes occur.
- People make mistakes.

If we can't build software in a strictly top-down manner, then how do we build software? Guindon (1990) studied how software developers think about problems. She presented a small set of software developers with a problem in designing an elevator control system for a building. She then videotaped and observed the developers as they worked through the problem. What she found was that the developers did not follow a top-down approach at all. Rather, the developers followed an *opportunistic* approach in which they moved freely between considering the requirements to inventing and discussing usage scenarios, to designing at various levels of abstraction. As the developers perceived opportunities to benefit from shifting their thinking between, they readily did so.

Stories acknowledge and overcome the problems raised by Parnas and Clements. Through their heavy reliance on conversation and their ability to be easily written and rewritten at varying levels of detail, stories provide a solution:

- that is not reliant on users fully knowing and communicating their exact needs in advance
- that is not reliant on developers being able to fully comprehend a vast array of details
- that embraces change

In this sense, stories acknowledge that software must be developed opportunistically. Since there can be no process that proceeds in a strictly linear path from high-level requirements to code, user stories easily allow a team to shift between high- and low-levels of thinking and talking about requirements.

User Stories Encourage Participatory Design

Stories, like scenarios, are engaging. Shifting the focus from talking about the attributes of a system to stories about users' goals for using the system leads to more interesting discussions about the system. Many projects have failed because of a lack of user participation; stories are an easy way to engage users as participants in the design of their software.

In *participatory design* (Kuhn and Muller 1993; Schuler and Namioka 1993) the users of a system become a part of the team designing the behavior of their software. They do not become a part of the team through management edict ("Thou shalt form a cross-functional team and include the users"); rather, the users become part of the team because they are so engaged by the requirements and design techniques in use. In participatory design, for example, users assist in the prototyping of the user interface from the beginning. They are not involved only after an initial prototype is available for viewing.

Standing in contrast to participatory design is *empirical design*, in which the designers of new software make decisions by studying the prospective users and the situations in which the software will be used. Empirical design relies heavily on interview and observation but users do not become true participants in the design of the software.

Because user stories and scenarios are completely void of technical jargon, they are totally comprehensible to users and customers. While well-written use cases may avoid technical jargon, readers of use cases must typically learn how to interpret the format of the use cases. Very few first-time readers of a use case have an implicit understanding of common fields on use case forms such as extensions, preconditions and guarantees. Typical IEEE 830 documents suffer from both the inclusion of technical jargon and from the inherent difficulty of comprehending a lengthy, hierarchically-organized document.

The greater accessibility of stories and scenarios encourages users to become participants in the design of the software. Further, as users learn how to characterize their needs in stories that are directly useful to developers, developers more actively engage the users. This virtuous cycle benefits everyone involved in developing or using the software.

Stories Build Up Tacit Knowledge

Because of the emphasis placed on face-to-face communication, stories promote the accumulation of tacit knowledge across the team. The more often developers and customers talk to each other and among themselves, the more knowledge builds up within the team.

Why Not Stories?

Having looked at a number of reasons why stories are a preferred approach to agile requirements, let's also consider their drawbacks.

One problem with user stories is that, on a large project with many stories, it can be difficult to understand the relationships between stories. This problem can be lessened by using roles and by keeping stories at a moderate to high level until the team is ready to start developing the stories. Use cases have an inherent hierarchy that helps when working with a large number of requirements. A single use-case, through its main success scenario and extensions, can collect the equivalent of many user stories into a single entity.

A second problem with user stories is that you may need to augment them with additional documentation if requirements traceability is mandated of your development process. Fortunately, this can usually be done in a very lightweight manner. For example, on one project where we were doing subcontracted development to a much larger, ISO 9001-certified company, we were required to demonstrate traceability from requirements to tests. We achieved this in a very light manner: At the start of each iteration we produced a document that contained each story we planned to do in the iteration. As tests were developed, the test names were added to the document. During the iteration we kept the document up to date by adding or removing stories that moved into or out of the iteration. This addition to our process probably cost us an hour a month.

Finally, while stories are fantastic at enhancing tacit knowledge within a team, they may not scale well to extremely large teams. Some communication on large teams simply must be written down or the information does not get dispersed among as many on the team. However, keep in mind the very real tradeoff between a large number of people knowing a little (through written, low-bandwidth documents) and a smaller number of people knowing a lot (through high-bandwidth face–to–face conversations).

Summary

- User stories force a shift to verbal communication. Unlike other requirements techniques that rely entirely on written documents, user stories place significant value on conversations between developers and users.
- The shift toward verbal communicate provides rapid feedback cycles, which leads to greater understanding.
- User stories are comprehensible by both developers and users. IEEE 830 software requirements specifications tend to be filled with too much technical or business jargon.
- User stories, which are typically smaller in scope than use cases and scenarios but larger than IEEE 830 statements, are the right size for planning. Planning, as well as programming and testing, can be completed with stories without further aggregation or disaggregation.
- User stories work well with iterative development because it is easy to start with an epic story and later split it into multiple smaller user stories.
- User stories encourage deferring details. Individual user stories may be written very quickly, and it is also extremely easy to write stories of different sizes. Areas of less importance, or that won't be developed initially, may easily be left as epics, while other stories are written with more detail.
- Stories encourage opportunistic development, in which the team readily shifts focus between high and low levels of detail as opportunities are discovered.
- Stories enhance the level of tacit knowledge on the team.
- User stories encourage participatory, rather than empirical, design, in which users become active and valued participants in designing the behavior of the software.
- While there are many reasons to use stories, they do have some drawbacks: on large projects it can be difficult to keep hundreds or thousands of stories organized; they may need to be augmented with additional documents for traceability; and, while great at improving tacit knowledge through face–to–face communication, conversations do not scale adequately to entirely replace written documents on large projects.

Developer Responsibilities

- You are responsible for understanding why you have chosen any technique you choose. If the project team decides to write user stories, you are responsible for knowing why.
- You are responsible for knowing the advantages of other requirements techniques or for knowing when it may be appropriate to apply one. For example, if you are working with a customer and cannot come to an understanding about a feature, perhaps discussing an interaction design scenario or developing a use case may help.

Customer Responsibilities

- One of the biggest advantages of user stories over other requirements approaches is that they encourage participatory design. You are responsible for becoming an active participant in designing what your software will do.

Questions

13.1 What are four good reasons for using user stories to express requirements?

13.2 What can be two drawbacks to using user stories?

13.3 What is the key difference between participatory and empirical design?

13.4 What is wrong with the requirements statement, "All multipage reports should be numbered"?

This page intentionally left blank