Advantages of User Stories for Requirements

- by Mike Cohn
- 40 Comments
- originally published in InformIT Network on 2004-10-08

Extreme programming (XP) introduced the practice of expressing requirements in the form of user stories, short descriptions of functionality–told from the perspective of a user–that are valuable to either a user of the software or the customer of the software. The following are typical user stories for a job posting and search site:

- A user can post her resume to the web site.
- A user can search for jobs.
- A company can post new job openings.
- A user can limit who can see her résumé.

But user stories are not just these small snippets of text. Each user story is composed of three aspects:

1. Written description of the story, used for planning and as a reminder
2. Conversations about the story that serve to flesh out the details of the story
3. Tests that convey and document details that can be used to determine when a story is complete

Because user story descriptions are traditionally handwritten on paper note cards, Ron Jeffries has named these three aspects with the wonderful alliteration of *card*, *conversation*, and *confirmation*.1 The card may be the most visible manifestation of a user story, but it's not the most important. While the card may contain the text of a story, the details are worked out in the conversation and then recorded and verified through the confirmation.

Why User Stories?

Because stories exhibit some of the same characteristics of use cases or traditional requirements statements, it's important to look at what distinguishes stories from these earlier requirements techniques. These differences can lead to many advantages for user stories.

Let's Be Precise

User stories emphasize verbal communication. Written language is often very imprecise, and there's no guarantee that a customer and developer will interpret a statement in the same way. For example, at lunch recently I read this on my menu: "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 act as though written words are precise, yet they often 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.

As another example, I recently came across this requirement, 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's unclear whether the user must enter a name for the folder. Perhaps a default name is provided. The second sentence is almost completely meaningless. Can the folder name be other lengths, or must it always be 127 characters?

Useful for Planning

A second advantage of user stories is that they can be used readily in project planning. User stories are written so that each can be

given an estimate of how difficult or time–consuming it will be to develop; use cases, on the other hand, are generally too large to be given useful estimates. Also, a story is implemented all in a single iteration of an agile project, while it's common to split a use case across multiple iterations (even though those iterations are usually longer than on a story–driven project).

IEEE 830–style requirements statements ("The system shall…") represent a different problem. 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's easy to see the inherent difficulty in prioritizing them. If the requirements cannot be prioritized beyond the common high, medium, and low, they're unsuitable for a highly iterative and incremental development process that will deliver working software every two to four weeks.

Spare Me the Details

User stories have additional advantages, but I'll provide only one more. User stories 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 detailed stories once it becomes important to have the details. This technique makes user stories perfect for time–constrained projects. A team can very quickly write a few dozen user 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.

User Stories Aren't Use Cases

First introduced by Ivar Jacobsen,2 use cases are today most commonly associated with the Unified Process. A use case is a generalized description of a set of interactions between the system and one or more actors, where an actor is either a user or another system. Use cases may be written in unstructured text or to conform with a structured template. The templates proposed by Alistair Cockburn3 are among the most commonly used. A sample is shown in the sidebar *Use Case 1*, which is equivalent to the user story "As a recruiter, I can pay for a job posting with a credit card."

Use Case 1

Use Case Title: Pay for a job posting.
Primary Actor: Recruiter
Level: Actor goal
Precondition: The job information has been entered but is not viewable.
Minimal Guarantees: None
Success Guarantees: Job is posted; recruiter's credit card is changed.
Primary Actor: Recruiter

Main Success Scenario

1. Recruiter submits credit card number, date, and authentication information.
2. System validates credit card.
3. System charges credit card full amount.
4. Job posting is made viewable to job seekers.
5. Recruiter is given a unique confirmation number.

Extensions

2a: The card is not a type accepted by the system.
2a1: The system notifies the user to use a different card.
2b: The card is expired.
2b1: The system notifies the user to use a different card.
2c: The card is invalid.
2c1: The system notifies the user to use a different card.
3a: The card has insufficient credit available to post the ad.
3a1: The system charges as much as it can to the current card.
3a2: The user is told about the problem and asked to enter a second credit card for the remaining charge. The use case continues at Step 2.

Within a use case, the term *main success scenario* refers to the primary successful path through the use case. In this case, success is

achieved after completing the five steps shown. The Extensions section defines alternative paths through the use case. Often, extensions are used for error handling; but extensions are also used to describe successful but secondary paths, such as in extension 3a of Use Case 1. Each path through a use case is referred to as a scenario. So, just as the main success scenario represents the sequence of steps 1–5, an alternate scenario is represented by the sequence 1, 2, 2a, 2a1, 2, 3, 4, 5.

One of the most obvious differences between user stories and use cases is their scope. Both are sized to deliver business value, but stories are kept smaller in scope because we place constraints on their size (such as “no story can be expected to take more than 10 days of development work”) so that they can be used in scheduling work. A use case almost always covers a much larger scope than a story. For example, looking at the user story “A Recruiter can pay for a job posting with a credit card,” we see that it’s similar to the main success scenario of Use Case 1. This leads to the observation that a user story is similar to a single scenario of a use case. Each story is not necessarily equivalent to a main success scenario; for example, we could write the story “When a user tries to use an expired credit card, the system prompts her to use a different credit card,” which is equivalent to Extension 2b of Use Case 1.

User stories and use cases also differ in the level of completeness. James Grenning has noted that the text on a story card plus acceptance tests “are basically the same thing as a use case.” By this, Grenning means that the story corresponds to the use case’s main success scenario, and that the story’s tests correspond to the extensions of the use case.

For example, the following might be appropriate acceptance test cases for the story “A Recruiter can pay for a job posting with a credit card:”

- Test with Visa, MasterCard, and American Express (pass)
- Test with Diner's Club (fail)
- Test with good, bad, and missing card ID numbers
- Test with expired cards
- Test with different purchase amounts (including one over the card’s limit)

Looking at these acceptance tests, we can see the correlation between them and the extensions of Use Case 1.

Another important difference between use cases and stories is their longevity. Use cases are often permanent artifacts that continue to exist as long as the product is under active development or maintenance. User stories, on the other hand, are not intended to outlive the iteration in which they’re added to the software. While it’s possible to archive story cards, many teams simply rip them up.

An additional difference is that use cases are more prone to including details of the user interface, despite admonishments to avoid this tactic. There are several reasons. First, use cases often lead to a large volume of paper, and without another suitable place to put user interface requirements, they end up in the use cases. Second, use case writers focus too early on the software implementation rather than on business goals.

Including user interface details causes definite problems, especially early in a new project when user interface design should not be made more difficult by preconceptions. I recently came across the use case shown in the sidebar *Use Case 2,* which describes the steps for composing and sending an email message.

Use Case 2

Use Case Title: Compose and send email message

Main Success Scenario

1. User selects the *New Message* menu item.
2. System presents the user with the *Compose New Message* dialog.
3. User edits email body, subject field, and recipient lines.
4. User clicks Send button.
5. System sends the message.

User interface assumptions appear throughout this use case: a *New Message* menu item, a dialog box for composing new messages, subject and recipient input fields in that dialog box, and a Send button. Many of these assumptions may seem good and safe, but they may rule out a user interface in which I click a recipient's name rather than typing it to initiate the message. Additionally, the use case of Use Case 2 precludes the use of voice recognition as the interface to the system. Admittedly, far more email clients work with typed messages than with voice recognition, but the point is that a use case is not the proper place to specify the user interface in this manner.

Think about the user story that would replace Use Case 2: "As a user, I can compose and send email messages." No hidden user interface assumptions. With user stories, the user interface will come up during the conversation with the customer.

To get around the problem of user interface assumptions in use cases, Constantine and Lockwood[4] have suggested the concept of essential use cases. An essential use case is one that has been stripped of hidden assumptions about technology and implementation details. For example, the following table shows an essential use case for composing and sending an email message. What's interesting about essential use cases is that the user intentions could be directly interpreted as user stories.

User IntentionSystem ResponsibilityCompose email message Indicate recipient(s)Collect email content and recipient(s)Send email messageSend the message

Another difference is that use cases and user stories are written for different purposes. Use cases are written in a format acceptable to both customers and developers so that each may read and agree to the use case. The purpose of the use case is to document an agreement between the customer and the development team. User stories, on the other hand, are written to facilitate release and iteration planning, and to serve as placeholders for conversations about the users' detailed needs.

Not all use cases are written by filling in a form, as shown in Use Case 1. Some use cases are written as unstructured text. Cockburn refers to these as use case briefs. Use case briefs differ from user stories in two ways. First, since a use case brief must still cover the same scope as a use case, the scope of a use case brief is usually larger than the scope of a user story. That is, one use case brief will typically tell more than one story. Second, use case briefs are intended to live on for the life of a product. User stories, on the other hand, are discarded after use. Finally, use cases are generally written as the result of an analysis activity, while user stories are written as notes that can be used to initiate analysis conversations.

User Stories Aren't Requirements Statements

The Computer Society of the Institute of Electrical and Electronics Engineers (IEEE) has published a set of guidelines on how to write software requirements specifications.[5] This document, known as IEEE Standard 830, was last revised in 1998. The IEEE recommendations cover such topics as how to organize the requirements specification document, the role of prototyping, and the characteristics of good requirements. The most distinguishing characteristic of an IEEE 830–style software requirements specification is the use of the phrase "The system shall…" which is the IEEE's recommended way to write functional requirements. A typical fragment of an IEEE 830 specification looks similar to the following:

4.6) The system shall allow a company to pay for a job posting with a credit card.
4.6.1) The system shall accept Visa, MasterCard, and American Express cards.
4.6.2) The system shall charge the credit card before the job posting is placed on the site.
4.6.3) The system shall give the user a unique confirmation number.

Documenting a system's requirements to this level is tedious, error-prone, and very time-consuming. Additionally, a requirements document written in this way is, quite frankly, boring to read. Just because something is boring to read is not sufficient reason to abandon it as a technique; however, if you're dealing with 300 pages of requirements like this (and that would only be a medium-sized system), you have to assume that it's not going to be read thoroughly by everyone who needs to read it. Readers will either skim or skip sections out of boredom. Additionally, a document written at this level will frequently make it impossible for a reader to grasp the big picture.

There's a tremendous appeal to the idea that we can think, think, think about a planned system and then write all the requirements as "The system shall…." That sounds so much better than "If possible, the system will…" or even "If we have time, we'll try to…" that better characterizes the reality on most projects.

Unfortunately, it's effectively impossible to write all of a system's requirements this way. A powerful and important feedback loop occurs when users first see the software being built for them. When users see the software, they come up with new ideas and change their minds about old ideas. When changes are requested to the software contemplated in a requirements specification, we've become accustomed to calling it a "change of scope." This type of thinking is incorrect for two reasons. First, it implies that the software was at some point sufficiently well-known for its scope to have been considered fully defined. It doesn't matter how much effort is put into upfront thinking about requirements; we've learned that users will have different (and better) opinions once they see the software. Second, this type of thinking reinforces the belief that software is complete when it fulfills a list of requirements, rather than when it

fulfills the goals of the intended user. If the scope of the user's goals changes, perhaps we can speak of a “change of scope,” but the term is usually applied even when only the details of a specific software solution have changed.

IEEE 830–style requirements have sent many projects astray because they focus attention on a checklist of requirements rather than on the userá¾¿s goals. And lists of requirements don't give the reader the same overall understanding of a product that user stories do. It's very difficult to read a list of requirements without automatically considering solutions in your head as you read. Carroll, for example, suggests that designers “may produce a solution for only the first few requirements they encounter.”6 For example, consider the following requirements:7

3.4) The product shall have a gasoline-powered engine.
3.5) The product shall have four wheels.
3.5.1) The product shall have a rubber tire mounted to each wheel.
3.6) The product shall have a steering wheel.
3.7) The product shall have a steel body.

By this point, I suppose images of an automobile are floating around your head. Of course, an automobile satisfies all of the requirements listed above. The one in your head may be a bright red convertible, while I might envision a blue pickup. Presumably the differences between your convertible and my pickup are covered in additional requirements statements.

But suppose that instead of writing an IEEE 830–style requirements specification, the customer told us her goals for the product:

- The product makes it easy and fast for me to mow my lawn.
- I am comfortable while using the product.

By looking at goals, we get a completely different view of the product: the customer really wants a riding lawnmower, not an automobile. These goals are not user stories, but where IEEE 830 documents are a list of requirements, stories describe a user’s goals. By focusing on the user’s goals for the new product, rather than a list of attributes of the new product, we can design a better solution to the user’s needs.

A final difference between user stories and IEEE 830–style requirements specifications is that with the latter the cost of each requirement is not made visible until all the requirements are written. The typical scenario is that one or more analysts spends two or three months (often longer) writing a lengthy requirements document. This document is then handed to the programmers, who tell the analysts (who relay the message to the customer) that the project will take 24 months, rather than the six months they had hoped for. In this case, time was wasted writing the three-fourths of the document that the team won't have time to develop, and more time will be wasted as the developers, analysts, and customer iterate over which functionality can be developed in time. With stories, an estimate is associated with each story immediately. The customer knows the velocity of the team and the cost of each story. When she has written enough stories to fill all the iterations, she knows she's done.

Kent Beck explains this difference with an analogy of registering for wedding gifts.8 When you register, you don’t see the cost of each item. You just make a wish list of everything you want. That may work for weddings, but it doesn’t work for software development. When a customer places an item on her project wish list, she needs to know its cost.

References

1. Ron Jeffries, “Essential XP: Card, Conversation, and Confirmation,” XP Magazine, August 30, 2001.
2. Ivar Jacobson, Object-Oriented Software Engineering (Addison-Wesley, 1992, ISBN 0201544350).
3. Alistair Cockburn, Writing Effective Use Cases (Addison-Wesley, 2000, ISBN 0201702258).
4. Larry L. Constantine and Lucy A. D. Lockwood, Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design (Addison-Wesley, 1999, ISBN 0201924781).
5. IEEE Computer Society, IEEE Recommended Practice for Software Requirements Specifications, 1998.
6. John M. Carroll, Making Use: Scenario-Based Design of Human-Computer Interactions (MIT Press, 2000).
7. Adapted from Alan Cooper’s The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How To Restore The Sanity (Sams, 1999, ISBN 0672316498).
8. Personal communication, November 7, 2003.

Posted: October 8, 2004

Tagged:

- user stories
- teams
- programming
- management
- customers
- velocity
- prioritizing
- use cases

About the Author

As the founder of Mountain Goat Software, Mike Cohn specializes in helping companies adopt and improve their use of agile processes and techniques to build extremely high-performance teams. He is the author of *User Stories Applied for Agile Software Development*, *Agile Estimating and Planning*, and *Succeeding with Agile*. Mike is a founding member of the Agile Alliance and Scrum Alliance. He is also the founder of FrontRowAgile.com, an online agile training website. He can be reached at info@mountaingoatsoftware.com or connect with Mike on Google+.