How Programmers and Testers (and Others) Should Collaborate on User Stories

by Mike Cohn •

57 Comments

Image not readable or empty
How Programmers and Testers (and Others) Should Collaborate on User Stories

Have the testers on your team ever struggled with what they should do early in a sprint before the programmers have finished coding a user story?

And have the programmers ever wondered what they should do near the end of the sprint while the testers are verifying the implemented code? Are they tempted, perhaps, to start on the next user story even though it can't be finished in the sprint?

When a team discovers the right way to collaborate on user stories, these questions vanish. Plus, the team is able to deliver more value more regularly to its stakeholders.

The Wrong Way to Collaborate on a User Story

Before we can identify the right way to collaborate on a user story, let's discuss the most common way teams collaborate on a user story--and why it's the wrong way.

The way most programmers and testers collaborate--if we can even call it collaboration--is for a programmer to grab a user story from the product backlog, fully code it to their own satisfaction, and then hand off the entire user story to a tester for validation.

When a programmer does this, though, the tester is left idle while waiting for the programmer to finish with the story.

And that's when testers begin to wonder what they should do with their time at the start of the sprint.

The Problems with Testers Working a Sprint Behind

One bad solution many teams may contemplate (or even try) is having testers work a sprint behind the coders. This creates a number of problems.

First, it stretches the amount of time a feature is in development. If the feature will be shipped when done, this means time to market increases. Even if the feature won't be shipped immediately, it now takes two sprints instead of one before the team can get feedback from stakeholders going hands-on with the feature.

Second, it also takes longer for programmers to receive feedback on the quality of their code. For example, if the new feature works but performs slowly, the programmers won't know it until the second sprint. If the team's new programmer isn't writing code to the same bug-free level of the others, that won't be discovered until the second sprint.

Third, by the time programmers do get feedback, the code is no longer fresh in their minds. I know it can be a cliche, but many years ago I was told there was a bug in my code, and I truly didn't even remember having written that code. The closer a defect is discovered to when it is created, the easier it will be for the programmer to fix.

The Nature of the Problem when Collaborating on a Story

The problem in how most team members collaborate on a user story is that the programmer finishes the entire user story before handing it over to the tester. This is essentially a large batch transfer. A big amount of something is finished by one person and then handed to the next person.

Have you ever sat down for a dinner in a restaurant and had a very large party of people order right before you did? If you're like me, when that happens, your immediate thought is that it will take longer than normal to get your meal. This is because a large batch (the other party's order) was just transferred from the server to the kitchen and yours will be behind it.

Just as large batch transfers can cause problems when you want to eat dinner, they cause problems when collaborating on a user story. To successfully collaborate on a user story, team members need to avoid large batch transfers.

How to Avoid Large Batch Transfers

The way to avoid large batch transfers is for a programmer to transfer small parts of a user story to the tester.

I recommend this begin with a programmer and tester talking about what part of a story to develop first. Once that's decided, the programmer goes away to code just that small portion. Simultaneously, the tester identifies the test cases, creates test data, and, if possible, automates the tests for just that part of the story.

When done, the programmer hands the implementation to the tester. The tester finishes any work on the automated test cases and runs the tests. Bugs are identified and, once resolved, the new feature is fully integrated into the system.

And then the process repeats: The programmer and tester discuss what part of the story to do next, do just that, and so on until all parts of the story are complete.

This is an incremental approach to the story. Work is done concurrently with frequent, small handoffs each time the programmer finishes a piece of the story.

An Example of a Programmer and Testing Working Together

Let's take a look at an example of how this might work. For our example, suppose the feature to be developed is a typical website login. The programmer and tester discuss where to begin. They could decide to begin with logging in through Facebook. Or maybe with requiring a strong password. Or maybe with allowing a user to request a password reminder.

For our example, let's assume the programmer and tester agree to start with the user entering a name and password and only being granted access if the two correspond. After agreeing this is the best place to start, the programmer codes it. While that is happening the tester designs the test cases. Once the programmer is done, the tester verifies the small piece of functionality

using the tests that have been created so far. Any issues are resolved.

With this small piece of functionality done, the programmer and tester meet again to discuss the story. This time they decide to add support for requiring a strong password. They probably need some guidance on how to define *strong* from the product owner or perhaps a security expert within a company. So together they go talk to the right person. Through these conversations, they determine that passwords must be at least 8 characters, contain at least one digit, one uppercase letter, one lowercase letter, and a symbol.

Equipped with this information, the programmer writes the code that will enforce these rules whenever a password is set. While the programmer is doing that, the tester is creating test cases that will validate each of these new rules.

When both are done, the code is tested and they move on to the next bit of the story, which is perhaps requesting a password reminder.

I've described this as a series of conversations. But there's no reason that the programmer and tester couldn't batch some of those conversations. They might, for example, decide everything I've just outlined in one conversation. If so, however, I'd still want the programmer to shout out, "Hey, Tester, I finished the first piece. Go ahead and test it." In some cases, it can be beneficial to decide what the next few steps will be in advance. This is fine as long as the handoffs remain small.

Can We Achieve the Same Result with Super-Small Stories?

As you read this example, you may have wondered, "But can't we achieve the same result by using super-small stories?" That is, couldn't we have stories like these:

- As a user, I'm only allowed to log in if I enter the correct credentials.
- As a user, I'm required to enter a strong password.
- As a user, I can request a password reminder.

Yes, you largely could achieve the same result. And while I generally favor fairly small stories, there are a few drawbacks to extremely small stories.

First, when working with overly small stories, there are more dependencies to manage. For example, if logging in is split into 20 stories, there will be dependencies to manage among all

those stories. If instead the team had a single story that fully encompassed logging in, there would be no dependencies to track, at least within the log-in subsystem.

Managing dependencies among many stories can become an issue; it is entirely possible that one story will become “lost” or separated from the others. When that happens, an important bit of functionality may be overlooked and not delivered.

A second difficulty caused by extremely small stories is that prioritizing the product backlog becomes more challenging and time-consuming. With more stories, it will take longer to sequence items on the product backlog. In many cases, that extra effort is wasteful because most or all of the small stories are the same priority and therefore adjacent on the ordered product backlog.

Finally, the more stories a team works on in a sprint, the more time will be spent tracking them and updating them in whatever tool the team uses. I would never opt for larger stories solely because of this, but it is something to be aware of.

So, small stories are generally good, but stories that are too small come with their own problems.

How small is too small? Look for the three problems I’ve outlined above. When you start to see those problems affect your team, your stories are starting to be too small.

This Applies to Handoffs Between any Roles

While I’ve used an example of a programmer and tester working together, it is important to note that any roles on an agile project can benefit from collaborating as I’ve described it.

Whenever two (or more) roles need to work together, they will do so most effectively by identifying small portions of the work and working concurrently to the extent possible. Reducing the size of handoffs from one role to the next is vital to effective collaboration between team members.

When working in this way, testers don't wonder what they should test early in a sprint. They test the initial small pieces handed to them by programmers. Similarly, programmers aren't left at the end of the sprint wondering if they should work on the next sprint while waiting for testers to run the first tests of the sprint.

What's Your Experience?

How is work handed off between members of your team? What other techniques do you use to encourage collaboration when working on user stories? Please share your thoughts in the Comments section below.

Posted: September 14, 2017

Tagged: user stories, teamwork, testing, collaboration, tester, programmer

---

About the Author

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 as well as the Better User Stories video course. Mike is a founding member of the Agile Alliance and Scrum Alliance and can be reached at hello@mountaingoatsoftware.com. If you want to succeed with agile, you can also have Mike email you a short tip each week.
