BPTrends COLUMN

THE AGILE PRACTITIONER TOM BELLINSON
July 2020

Agile Workflow

In this Column, I'd like to talk about two things that always seem to come up in software development, but they can most definitely apply to other product development processes as well.

1. The amount of work in progress
2. Coupling/interdependency of components

What is "Done?"

You can't talk about agile workflow without first being clear about your definition of "done." Depending on your particular workflow, the meaning can be quite different. However, since agile principles are all about delivering value to the customer, the closer you can get to that, the better.

In our case at ITHAKA, we have achieved a continuous delivery model of software development. This means that each developer can write some new code, deploy it to our test environment, run automated tests, and when those tests pass, push the code directly to our production environment which is for our millions of users.

Because of this, for us, "done" means that users are (or could be) using it. However, if you are designing and building car engines, "done" might need to be a step or two before that engine is in a vehicle that a consumer could use. In this case, what is important is that the engine could be used in a vehicle in its current state. Maybe you're only making the fuel injection system. In that case, you may say that "done" is when the system could work on a production engine.

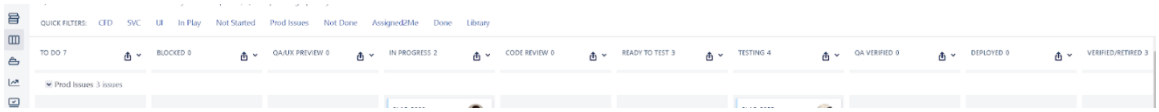
The point here is that you must be conscientious about your definition of "done" and use it to evaluate your workflow. Since my experience centers around software, my examples will use that milieu. Hopefully, if you build other things, you can find parallels to apply to yours.

What's In Progress?

The challenge that we constantly face is managing how much work we have going on at any given point in time. As a cross-functional team, we often battle the trade-offs between individual productivity and team productivity. On the surface, one might assume that maximizing individual productivity will be best for overall team productivity. We have learned through experience that this is often not the case.

To illustrate, let's look at the workflow as defined by one of my teams using Jira (a popular Atlassian product for managing software development workflow).

In case that graphic is hard to read, our workflow looks like this:



ToDo -> QA/UX Preview -> In Progress -> Code Review -> Ready To Test -> In Testing -> QA Verified -> Deployed -> Verified/Retired

We also have a step called "Blocked" that we use from any of the active steps to tell us when we cannot continue working due to factors external to the team.

What is important about this workflow is that different members of the team will be involved at various stages of the process. In the QA/UX Preview stage, our quality assurance leader on the team will review the instructions with the developer who is picking up the work to make sure there is understanding about the definition of a successful outcome and how it will be tested. If the particular work in question involves a user interface element, our product designer will also preview the work with the developer, often reviewing wireframes or models to ensure form and function are well understood before coding begins.

At the Code Review stage, another developer will review the developer's work on the task. By the way, Jira generically calls these "tasks" issues, but we often refer to them as user stories, since in the agile world individual work items tend to center around delivering specific value to users. In our case, code may even be reviewed by a developer who is not on our team. This essentially expands our definition of the team to include all of the other front-end development teams that share work of designing and developing the user interface for our system.

When a user story becomes Ready To Test, our QA person on the team will begin working with the developer to get it deployed to the testing environment and get it tested. At this point, our QA person may have already developed an automated testing suite, or he may need to do that at this stage if he hasn't had the chance to do it ahead of time.

A note about automated testing: while there are parallels in other manufacturing disciplines, in software development, automated testing is relatively new and very powerful. The reason is because it allows developers to rapidly make small changes and test them thoroughly without an extensive manual review. It also allows us to incrementally update the automated testing suites to reflect new or altered functionality such that we avoid regression (a topic that may get its own article at some point).

Once the work has successfully passed the test phase, it is deployed to the production environment and the QA person will quickly rerun all the tests just to make sure that it is working the same in production as it did in the test environment.

As you can see, different people are involved at various steps in our process. Usually, the bulk of the work is done by the developers. We have multiple developers on each team, but only one person for the other roles. This means that it is possible for developers to overwhelm other members of the team.

Developers can even run into each other. Because we have a continuous delivery pipeline, anyone may want to deploy changes to the same part of the site. Since nobody has figured out how to make simultaneous multiple changes to a component at this point, developers who have code ready to go will need to either wait, or merge their code with the other developer's code (more on this later).

When delays like this or waiting on other members of the team occur, it is tempting for developers to pick up more work, so that they can remain productive. Of course, this means that other members of the team will need to preview their new work, which takes them away from helping to complete work that is already in play.

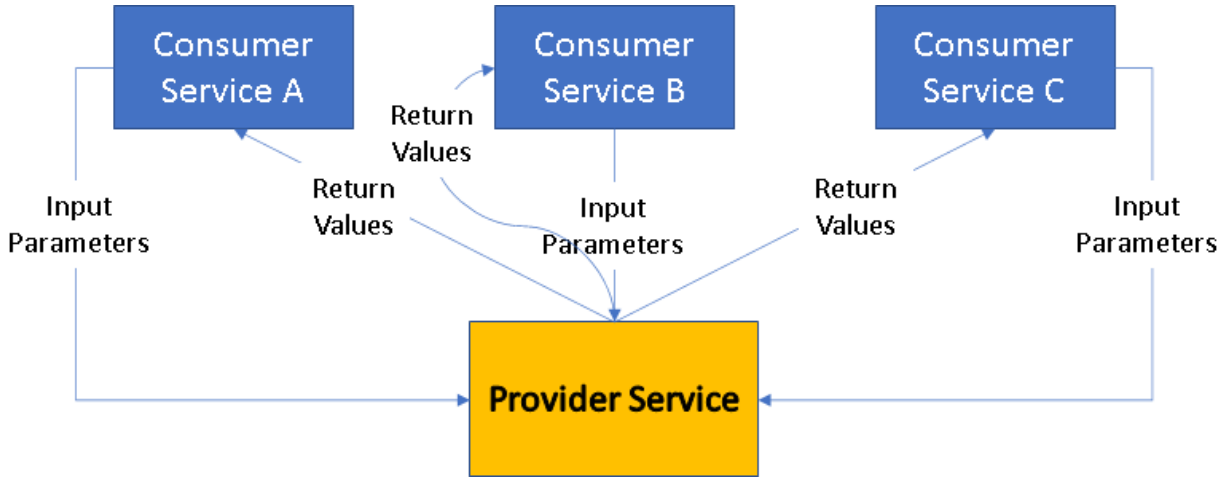
To mitigate the temptation to pick up new work, we encourage developers to help each other with the testing and deployment of user stories developed by other team members. We constantly remind ourselves that completing the intermediate steps of the process delivers no value to our users. Nonetheless, developers regularly pick up more work and we regularly run into the problems I've described. Some lessons are really hard to learn.

Coupling/Interdependencies

ITHAKA once used a 3rd party product that we purchased for our platform. When we decided to reimagine it with a system we built ourselves, we wanted to take an approach that would allow us to maximize our opportunities for incremental improvement.

To that end, we opted to use something called "micro services architecture." As the name suggests, this approach utilizes small single or minimal purpose programs that interact with one another to provide complex functionality. Each program can have information passed to it from other services and it can pass out information to other services.

When a program calls another program (or service) to ask for information or data, it needs to know what information the program it is calling needs to know in order to return the request. This "knowledge" must be contained within the calling program or "consumer" as we often refer to it. The challenge here is that the program that provides the data to the consumer (called the "provider") can change and potentially break the consumer program.



This problem gets quickly compounded when provider applications service multiple consumer applications. When a single provider must service multiple consumers, it must often have additional complexity to deal with the variations in output needed for each consumer. This complexity creates hazards that have caused us to question the concept of reuse, which has traditionally been very popular. The argument in favor of reuse is that you can build a set of common components that can serve the same purpose across the system. For example, if you want a feedback widget that pops up on every page in the site, you could choose to write one of these which can be called from any page and it will work the same. This idea seems like a good one at first.

Continuing with this example, let's say that for one of the pages which calls the feedback widget, the designer decides that she wants the color to be different. No problem says the developer. She just writes a new feature into the shared feedback widget that allows the consuming page to pass in a color value. This works great for the page, but suddenly all the other pages don't work with it because they're not passing in a color. A problem like this can be easily fixed by setting a default color for when one isn't provided. This is just an example of how complexity can get us into trouble and while some issues are easy to troubleshoot, the more interdependent and large a network of applications becomes, the more likely that developers will experience troubleshooting challenges.

The alternative is to write a separate provider whenever a significant variation is desired. This has the benefit of reducing both the complexity of the provider and its number of dependencies. While it does seem to add to the complexity of the overall system in that there will be a lot more of these small micro services, it makes changing them over time a lot easier and improves system stability.

Finally, the workflow benefit of having more micro services that are tightly coupled between consumer and provider, the less likely that developers will be slowed down by having to wait for other developers to "get out of their way" because they have work in progress in the same application.

Working in parallel is not always possible even in the best of circumstances, but the more the architecture of the system can facilitate it, the more efficient the team will be.

In Summary

Modern product development teams work together to deliver value to customers. Individuals who try to get ahead of the other members of the team rarely are able to deliver any extra value. Therefore, the focus needs to be on the throughput of the entire team. If one person becomes a bottleneck, other members of the team should be trained to step in and help with the work at the choke point. It's actually better for them to find other things to do rather than pile up work that cannot be completed by the team.

Further, good system architecture can contribute greatly to how efficiently a large complex product can be continually enhanced. By reducing interdependencies and complexity of individual components, the overall volume of components may grow, but the ability to make changes without breaking other parts of the system is greatly improved. This will ultimately help the performance of the product development team and THAT puts increased value in the hands of users more quickly.

Isn't that what we're all trying to do?

AUTHOR

Tom Bellinson

Mr. Bellinson has been working in information technology positions for over 30 years. His diverse background has allowed him to gain intimate working knowledge in technical, marketing, sales and executive roles. Most recently, Mr. Bellinson finds himself serving as a Scrum Master for ITHAKA, a global online research service. From 2008 to 2011 Bellinson worked with at risk businesses in Michigan through a State funded program which was administered by the University of Michigan. Prior to working for the University of Michigan, Mr. Bellinson served as Vice President of an ERP software company, an independent business and IT consultant, as chief information officer of an automotive engineering services company and as founder and President of a systems integration firm that was a pioneer in Internet services marketplace. Bellinson holds a degree in Communications with a Minor in Management from Oakland University in Rochester, MI and has a variety of technical certifications including APICS CPIM and CSCP.