

Best Practices of Agile Tool Users

Author:

Damon Poole, Chief Technology Officer

AccuRev

Best Practices of Agile Tool Users

You've decided to transition to Agile development. Everybody has been to Certified Scrum Master training, you've retained an Agile coach, you've got a Product Owner, a Scrum Master, and two week iterations. You've even switched to a new SCM system and acquired your first Continuous Integration and Agile Project Management tools. Now what? Are you worried that you have retained some non-Agile baggage and may not be using your Agile tools to their best advantage? That's a perfectly valid concern. Here are ten tried-and-true practices for Agile tool users which will get your started on your way towards optimal use of your Agile tool stack.

Evaluate the Agility of Your Tool Stack

In order to practice Best Practices for Agile tools, you have to have Agile tools. Thus a first step is to make sure that you have an Agile tool foundation and actively work to maintain an Agile tool stack. An Agile tool is a tool that supports one or more Agile practices and does so in a way that fits into the overall Agile framework. For instance, a tool which allows you to maintain a ranking of all of the issues that you care about is one that supports the Agile practice of maintaining a backlog. Agile tools must support a high ratio of value to effort in order to fit into the short iterations of an Agile project.

For each of the tools in your stack consider the following questions:

- How long ago did we implement this tool?
- Do we have the most recent version of this tool?
- Does the vendor of this tool use Agile development to produce this tool?
- What are the specific Agile practices that this tool supports?
- What are the specific features supporting those Agile practices?
- Are there other tools are better suited to those Agile practices?
- Are there Agile tools which remove or reduce the need for this tool?

For instance, if you are using Gantt charts, when you move to Agile development, you may be able to stop using Gantt charts and whatever tool you are using to produce them.

Measure Value Produced Instead of Progress Against Plan

A common metric on an Agile project is a burn-down chart. A burn-down chart shows on a daily basis the sum of the hours remaining for all work for the current iteration. The chart is used to make sure that there is regular progress. The problem is that you may end up getting 80% of the way through all of your assigned work, but still have nothing done. Doesn't this sound very familiar? It sounds exactly like one of the core problems that Agile is supposed to be solving, not perpetuating: measuring progress against plan.

One of the core claims of Agile is to measure value via working software produced. It therefore makes sense to use a metric which aligns with this core value. It is much better to be getting work done on a regular basis and then have a couple of items which are not done at the end of the iteration and get deferred to the next iteration rather than having unfinished work at the end of the iteration. This is nothing less than fine-grained application of the Agile principles.

A more Agile-oriented metric for measuring progress is the burn-up chart. A burn-up chart shows on a daily basis how much work is done and could be shipped. Now you can see if work is getting done on regular basis. If not, you know right away instead of at the end of the iteration.

On a side note, be aware that another problem with a burn-down chart is that it relies on developers entering how many hours remain on their tasks on a daily basis. Many developers see this as incongruous with the fact that they are salaried employees and it can be a daily source of irritation for them.

Maintain a Master List of All Work Items

In order to facilitate the use of a backlog, you really need to have all work represented in that backlog. Ideally, all work that is performed should be in the form of user stories and defects and managed via the backlog. Thus, all user stories and defects should be available via your Agile Project Management system. That doesn't mean that they need to be originated there, you may have a separate system for defect management, but all planned defects should be migrated into your Agile Project Management system. The benefit is that nothing falls through the cracks and all work can be managed, tracked, and measured via the same process.

Use Change Packages to Link User Stories and Defects to Source File Changes

There are many different methods for linking the work requested by the business to the work done by the development organization to fulfill that request. An all too common method is to have no linkage at all. Other forms of integration include typing a bug number into a comment, tight integration on a file/version level via scripting or built-in capability to do validation and other actions, tight integration via transactions (sometimes called change sets or change lists), and tight integration via change packages. Change packages are the best form of integration in general and also the best suited for Agile development.

A change package represents a complete patch of all changes required to implement a particular change. It is a rollup of all of the changes that have been applied to resolve an issue including a complete audit trail of who made the changes, when they made the changes, and why they made the changes. Each change package is tied to an issue in the Issue Tracking System (ITS).

Unlike other changed-based integrations where all information about the changes associated with an issue are stored in the ITS, change packages store all of the required information directly and the ITS information serves as a convenience to the ITS user. The advantage to using change packages is that the progress of the issue and its current state can be seen via the source control system which is a more natural interface for developers.

Change packages simplify the process of backing out changes. Any change can be backed out at any stage of the development process. If the change is being backed out due to a testing failure, it is easy for a developer to pick up work on the change and then reintroduce it when it is finished.

Fail Fast: Break Up Monolithic Builds and Test Suites

In Agile, feedback should be as quick as possible. Consider a typical "nightly build." In many shops, there are three kinds of builds: official builds, nightly builds, and developer builds. The official builds are done infrequently and are the builds that become the official product or officially deployed version of the software. Nightly builds are similar to the official builds, but are used for the sole purpose of providing feedback to development as to the state of the mainline. Developer builds are the builds that developers do for themselves as they do their work.

The official build of a piece of software can be a fairly complicated affair. The build script may consist of 100 or more steps, each of which may fail. Following the build, the test phase may involve the running of thousands of individual automated test, each of which may also fail. Typically, there is a check after the build to see whether or not to run the tests, but often the build and test phases are monolithic processes

and are not broken up into pieces. In an Agile environment, there will be multiple teams, each of which need their own build. The best situation is to be able to fail fast. That is, build the pieces of the software which have changed first, run the tests which are most likely to detect systemic failures first, etc. If you think of the process of going from source code to ready-to-install bits as the whole build process, you have a dependency graph of many steps which can be run in parallel across multiple machines. In order to do testing, it is also likely that you will need to use virtual machines in order to automate the fast setup and teardown of test environments. Yes, you can stick with your existing build and test environment and write scripts for this, but there is really no substitute for a modern build automation system.

Use Branches to Smooth Out End of Iteration Activities

With short iterations, it is likely that there will still be some touch-up work to be done on the previous iteration when the next one starts. Perhaps QA has not finished evaluating the iteration, or perhaps somebody wants to do a demo but runs into a bug that needs to be fixed first.

By branching, you can start the next iteration without disturbing the previous one. If QA finds any problems, development can fix them on the previous iteration branch. At some point that branch will be declared "done." Those changes are then merged into the current iteration. At that point, there is very little if any difference between where things are and where they would have been if development had started from the "done" baseline.

The advantage here is that development did not have to stop, and you still have a stable baseline. My experience is that once a team gets into a rhythm, the amount of time spent on "touch-up" is minimal.

This also applies to iterations that become releases. The "previous iteration" branch becomes work towards the release and then hopefully soon thereafter the release itself. Meanwhile, work continues towards the next release.

Use Multiple Tracks of Development to Work Towards Short Iterations

When first starting Agile development, it can be difficult to imagine, much less implement, short iterations for all work. One way to ease the transition is to divide your development tasks into two categories: those that easily break down into small user stories, and those that don't. Now you can do your development on two tracks instead of just one. The main track is work that can be completed within the iteration including all testing. The secondary track is for all other tasks. Once a secondary task has been completed, it gets merged into the primary track at the start of the next iteration. That way, most of your work is done with the same rhythm and you start to get into the habit of breaking stories down. The more you break stories down, the more opportunities you'll see to do it and the more skills you will build up to do it.

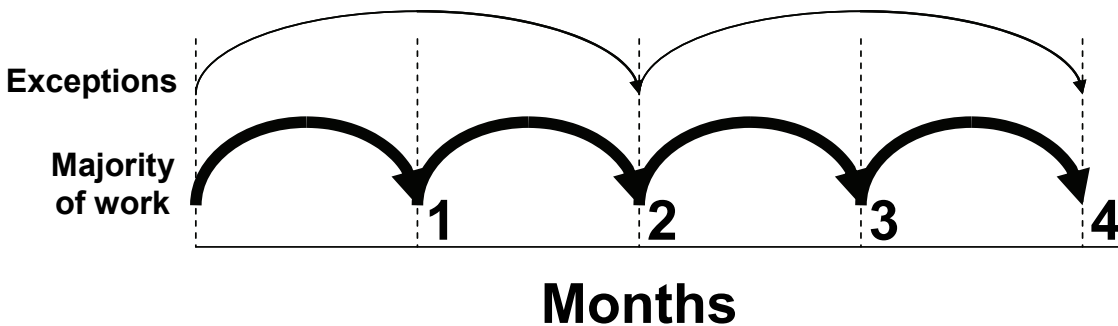


Figure: putting different kinds of work into different tracks.

After a few iterations, I think you will find that more and more stories will be able to be done on the main track and fewer and fewer will need to be done on the secondary track.

Use Permissive File Access Policies to Support Collective Code Ownership

SCM tools often have multiple modes for interacting with files, such as pessimistic locking, optimistic locking, etc. This usually also includes read-only file modes. The problem with some of these modes is that they interfere with an Agile practice known as Collective (or Shared) Code Ownership. To support Collective Code Ownership, use optimistic file locking (or no file locking at all), and set all file permissions to writeable. There should be as few barriers as possible to making file changes.

Base Iteration Reviews on SCM Baselines

At the end of every iteration, there is an iteration review. An iteration review is simply a demonstration of each user story to the product owner to show that the work has in fact been done. But how do you really know that what is being shown has been integrated into the mainline and passes all unit tests and other end of iteration criteria in accordance with the principles of Continuous Integration? Simple. The product owner should require that the team show that what they are demoing is a single set of binaries built from the same baseline and that baseline is from the mainline and built on an official build machine. This will validate that what is being shown has not been hacked together simply to get through the end of iteration review but does not actually meet the end of iteration criteria.

Mirror Your Agile Workflow with Branches or Streams

All software projects track the progress of the work to be done using some set of workflow stages. It may be a very complicated workflow that can fill multiple pages, or in the case of an Agile project it may be as simple as “reported,” “backlog,” “todo,” “in progress,” “coded,” “done.” People are used to keeping track of this status on a per-work-item basis in their issue tracking systems and Agile project management systems, but this does not generally translate over to their SCM system. In most cases, the SCM system models all states with a single concept: “the mainline.”

By mirroring your Agile workflow with branches (or streams), you can get many of the same benefits that tracking state gives you in your change management system in your SCM system. For instance, if a tester wants to do exploratory testing, they can query the change management system to find out which user stories for the current iteration are in the “coded” state. That way, they only test things which developers have indicated are ready for testing. If you use branches (or streams) to model your workflow, that same tester can go to the “coded” branch to check out the sources, build the system, and do exploratory testing on a version of the system containing only “coded” changes instead of including “in progress” changes.

Conclusion

Since you invested so much to get Agile training and acquire and install Agile tools, why not get the most out of your investment? By following these Best Practices you will increase your chances of success with Agile and increase the quality of the software you create with your Agile process.



AccuRev, Inc.
10 Maguire Road
Lexington, MA 02421

Phone: 1-800-383-8170
sales@accurev.com
www.accurev.com