



The Team That Walked Through Walls

Overcoming Barriers to Agile SCM

By Steve Berczuk

Steve is a software developer, author, trainer, and consultant. His areas of experience include building software systems using object-oriented languages and agile development techniques, software and organizational patterns, and software configuration management patterns and techniques. He works at [Fast Search and Transfer](#).

"... Pixel got the tag 'Schrödinger's Cat' hung on him because he walks through walls.'

'How does he do that?'

Jane Libby answered, 'It's impossible but he's so young he doesn't know it's impossible, so he does it anyhow.'"

-- [The Cat Who Walks Through Walls](#)
Robert Heinlein.

The Art of the Possible

One of the fundamental principles of Scrum is "The Art of the Possible." All agile methods have at their heart the idea that, if you rethink your work style and overcome the obstacles that old ways of thinking create, you can improve the feedback loop among and

improve value for project stakeholders. When teams start to explore agile methods, team members often believe that these processes can't possibly work because agile methods require people to do more than they possibly can. What is missing in these cases is the understanding that the current ways of doing things are what place constraints on what is possible. Very often teams don't do some simple things that can help them be more agile because they think that they cannot. One of the areas where this is often the case is the build process.

By improving your build process your team can deliver better quality code more quickly, and make the first steps towards being more agile.

But first they have to overcome their preconceptions of what they can and cannot do.

Builds and Feedback

An essential aspect of agile processes is the principle of feedback. You do something, you compare it to where you want to be, and you adjust. Agile methods do this at many levels:

- Between iterations, you compare a feature list to what you have planned for the next iteration.
- During an iteration, you compare where you are in relationship to your goal and adjust your work. In the developer workspace, you code and then run tests to see if your code meets the requirements specified by your tests.



An essential aspect of this feedback process is *working software*. To make any kind of effective measurement of how much you have done, you need consistently built software. And to maintain velocity you need to detect problems as soon as they occur. Needlessly broken builds stop the team. And inconsistencies between build environments can mask problems.

Patterns for Repeatable Builds

The first thing that a team can do to address these issues is to define and use a repeatable build process. There are [patterns](#) that will help your team's configuration management and build process be more agile. Four key patterns for this are:

- *Integration build*, which allows you to verify that developer changes work in a standard environment.
- *Private workspace*, which enables you to create developer environments that are consistent across team members.
- *Repository*, which provides a standard mechanism for getting the artifacts you need to create a private workspace.
- *Private system build* creates a

mechanism where a developer can build the software in a way that is close to the integration build environment. This removes problems caused by inconsistencies between a developer workspace and the standard workspace

One of the keys to an effective build process is that you create an *integration build* and a similar *private system build* in a *private workspace*.

A repeatable integration build process is one in which you can create a workspace anywhere and know exactly what you are building: what code, what versions of what libraries, etc. This is essential both for supporting customers and for avoiding wasting time.

A private workspace and private system build help to ensure that developers can test their changes before committing changes to a build, thus reducing the chance for a broken integration build.

I'll discuss some ways that you can move your team towards this goal in a bit, but first let me go through some of the common excuses teams use to avoid changing how they work.

Excuses, Excuses

It's hard to think of a reason to

not like reproducible builds and few argue that that they are a bad thing. Some people argue however that they are not right for their team at the moment. Here are some common rationales and counterarguments. Each of these excuses shares a common theme: inertia is often difficult to overcome when you have a system that works adequately.

- *We have something that works, why change it?* If the system you have in place does, in fact, work well, they maybe you don't need to change a thing. In many cases build processes that are not repeatable and executable in both an integration and development workspaces work only in the sense that you can build an executable of some sort. If you find yourself with frequent broken builds or with problem reports that are caused by an incorrect version of some artifact then what you have doesn't work and you should change it.
- *We don't have the time to change now; we have a release to get out the door.* Reworking the build process can be an incremental process. You can pick a couple of modules to experiment with and



adapt the process to the others. If you really can't take any kind of hit in your schedule, then you can start a parallel process in the background. If you are working off of the same codeline you can create a parallel integration build process and run all of your unit and integration tests against that as well as your current build. You do have integration and unit tests, right?

- *I don't need to setup/run a build in my workspace. I'm a developer.* There are two sides to this issue. It's true that not everyone on a team needs to know the inner workings of the build and release toolset. Even on agile interdisciplinary teams there will be experts on certain aspects of your application. However, everyone should be willing and able to pitch in. You should create a process that is easy to execute and that requires little day-to-day work, but the build should be something that everyone owns, much as testing is.

There are many more reasons one can come up with to not improve the build process. They have their roots in a natural tendency to keep things

as they are because they work well enough. The risk is that the way that things work now is often optimized for the short term and the nearby. The fact that a process works well for the development team and allows them to deliver code to a QA team, for example, does not mean that you are delivering a better quality product, or that the end-to-end delivery time is shorter. Often a locally optimized process creates a situation where you have more cycles because it is harder to identify and reproduce configurations down the line. As agile developers, we need to focus on delivering value to stakeholders and processes that make it easier to locate problems earlier, increase feedback, and increase the value.

Some of the changes you need to make are difficult, but many are not as daunting as they seem once you get started.

Better Builds

Developers on agile teams need a way to set up their workspaces quickly and easily. Enabling your team to have consistent builds will allow you to have a more accurate picture of the real status of the code and avoid extra cycles that having inconsistent environments can cause. Having a

well-defined workspace creation process will also allow new developers to become productive more quickly—setting up a development environment should take minutes, not days.

A side effect of having a good workspace creation process is that you are now doing builds on every developer's machine many times a day. Doing this will allow you to identify configuration issues, as it is inevitable that each developer will have slightly different configuration parameters. For example, you might have a test that requires a database; developers will have different database connections. Or you might need to access a file system resource; having every developer build and run a suite of tests makes it easy to detect when you've been relying on some sort of hard-coded resource path.

You can create a process that has the following steps:

- Install some basic tools, for example a version control client, a compiler, and perhaps a build tool such as [Maven](#) or [ant](#). Installing these tools can be part of a documented, manual process, or you can reduce this step to one by running a script that installs everything for you.

- Checkout a project from version control containing all of the source you need for your project. If you are using a tool like Maven, dependencies will be received automatically from a central repository.
- Build the project. A tool like Maven or ant will reduce this to one line.

Once you think about how your project is set up, this is a straightforward process to implement. The hard part is that making your workspace creation process standard and reproducible will force you to confront inconsistencies in your approach. But it's better to find the inconsistencies now than once your application is deployed at a customer site.

The exact details of how to create a workspace will vary from project to project. The requirements for the process are:

- Steps that are repeated frequently should be automated to reduce the chance for errors. You want your development team to focus on writing and testing code; you don't want them spending cycles on remembering manual processes,

While it might be acceptable to ask team members to download some files to set up a project, the regular builds should be automated. If there *are* manual steps, however, you should have some mechanisms in place to identify when a manually-installed component is out of date.

- The build process should be reproducible. If you were to reproduce the process on a new machine you should get the same results.

Now your developers can build a workspace easily, and in the same way as everyone else. This makes it easier to get reproducible results.

Summary

Fixing your build process will provide for huge rewards. Fixing the process may uncover other issues with your infrastructure. Be prepared to address them and you will more than make up time spent in improving how you work. Making these changes may seem difficult, but, like many aspects of adopting an agile process, the hardest part is overcoming the tendency to not want to change.



The Secrets of Agile Teamwork:

BEYOND TECHNICAL SKILLS



December 5-7
Portland, Oregon

- Improve the quality of interactions with team members, customers and others outside the team
- Increase the speed and effectiveness of feedback
- Contribute to an environment for team success

How do you develop, grow, and maintain a functioning self-organizing team?

Effective self-organizing teams rely on personal and interpersonal effectiveness. In this workshop, we'll practice the skills you need to succeed and lead on a self-organizing team.

Spend three days with two of the field's most effective creators of high-performing teams — Esther Derby and Diana Larsen.

FOR MORE INFORMATION, CONTACT:

Esther Derby

derby@estherderby.com • 612-724-8114

Diana Larsen

dlarsen@futureworksconsulting.com • 503-288-3550

Download a registration form at:

<http://www.estherderby.com/downloads/SATDec2006.pdf>