If It Doesn't Run - It Ain't Agile

By Bob Bretall
Enterprise Agile Coach

Integrating code and making sure it runs correctly can be tough. That is exactly why it should be done regularly (at least daily) throughout each iteration and not put it off until the end of the iteration. This is commonly called ***continuous integration***. A definition I like for continuous integration comes from Martin Fowler:

A fully automated and reproducible build, including testing, that runs many times a day. This allows each developer to integrate daily, reducing integration problems.

A great goal, but one many Agile development shops are not realizing. The idea of continuous (or frequent) integration is not a new one, nor did it originate with Agile, but I would consider it to be a cornerstone of Agile software development. The converse to continuous integration is teams "doing Agile" who don't integrate and run their code until the tail end of an iteration, creating a mini-waterfall. This causes many of integration breakage issues that happen in waterfall oriented methodologies, albeit on a much more manageable scale. I have seen a lot of angst amongst team members trying to get a build to work when they do the initial build for an iteration on Thursday evening before a Friday iteration close-out and discover integration problems. Any team will eventually get their code to run. Reducing the effort to get it running is the goal of Agile and continuous integration.

THE DESARA GROUP

Any team will eventually get their code to run.

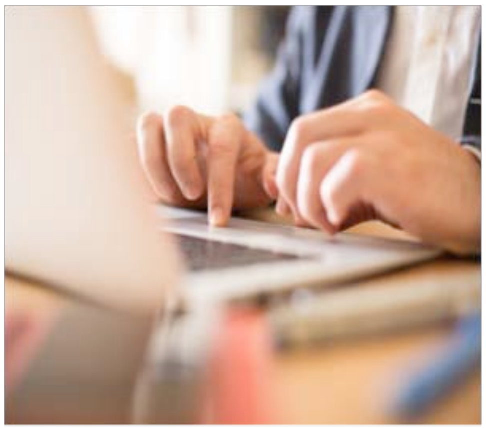
Reducing the effort to get it running is the goal of Agile and continuous integration.

The key is integrating, building and testing the software on a regular basis. Several times a day if possible. Different times of day and different frequencies will be right for different teams. The important thing is doing it on a regular cadence. A problem with a single nightly build is that issues will not be resolved until the next day and if all developers don't show up to work at the same time (which I have found to often be the case) you can have delays waiting for someone to show up in the morning to fix a problem.

Verifying each build with appropriate testing ensures integration errors are found and addressed as quickly as possible. Many teams believe that this approach to development leads to significantly less time spent on integration problems and allows for more rapid development of their software in general. Far from waiting months to integrate and run code, in Agile we don't want to wait even 2 weeks! Get it done multiple times a day!

Continuous integration was popularized as one of the core principles of Extreme Programming (XP), but the concept pre-dates XP. I was first exposed to the idea of continuous integration in Grady Booch's 1993 book "Object-Oriented Analysis and Design with Applications" (2nd edition).

I was big into OOAD back in the early 90s while working on my Master's degree in Computer Science. At the time Booch's book was a key text for me. I always felt that object-oriented analysis, design, and development were linked with iterative software development, probably because that was the way we did it on every OOAD project I worked on over the years. The concept of iterative development predates Agile and occupies a time in the 1990s between the heyday of Waterfall in the 1970s and 80s and the formal rise of Agile in 2001. Iterative development, refined over the years, is a big part of what makes Agile so powerful. But iterations have to be done right! Done wrong, iterative development can devolve into a series of 2 to 3 week waterfalls. I'm guilty myself of using the "iteration as a mini waterfall" analogy in years past in an attempt to make iterations easier for people transitioning off waterfall to understand and assimilate iterations into their thinking. I now know that it is far better to make a clean break and disabuse people of the notion that an iteration is a mini-waterfall. More properly, an iteration is a timebox during which development takes place with a goal of fully completing work assigned to the iteration.



Done wrong, iterative development can devolve into a series of 2 to 3 week waterfalls.

Mull that last part over: *fully completing work assigned to the iteration*. That means it needs to be built, tested and it **runs**. Why is this important to continuous integration? Continuous integration helps facilitate completing the work assigned to an iteration. I've often seen teams that wait until the end of an iteration to integrate run into problems at the end and struggle to fix them. Since an iteration is time-boxed, if they are unable to fix those late breakage issues found close to the end of an iteration, they tend to roll over into the next iteration waiting to be resolved. This means that work was not completed in its originally planned iteration but slips into the next. I have seen this create a rolling wave of work slippage from iteration to iteration until something changes. Implementing more frequent build/test/run moved us away from "build at the end of the iteration" towards continuous integration.

A key contributor to cost and schedule over-runs in software development is the late discovery of problems. The degenerative case of this is, of course, waterfall development where actual build/integration/test can happen many months after the project begins. Though many people "doing waterfall" are actually working in a kind of hybrid environment where they will begin coding earlier and do pseudo iterations or some other kind of phased development. There is a very large sliding scale of development with pure waterfall on one end and full-on Agile with continuous integration (supported by automated builds/tests) on the other. Late discovery of problems is the sickness, continuous integration is the cure.

Key Elements of Continuous Integration

There are lots of more detailed examinations of continuous integration than I'm presenting here, but I wanted to hit the highlights of things I've seen that help:

- **Code Repositories** – Lots of developers work on lots of files that all need to be coordinated to build a product. I've seen teams that try to keep things straight using shared drives and other manual forms of file sharing and it generally does not go well. Once you are working with more than a very small number of people, having a code repository to help you keep everything straight is an essential part of development. Get a source code management tool. There are open source choices (like GitHub or Subversion) as well as commercial choices like Perforce and ClearCase, among others. Make sure you get people who know what they are doing to work out a branching strategy for your development that will suit the needs of your team.

- **Build Automation** – Creating a running system can be complicated. Lots of files are involved that need to be compiled, database schemas may need to be loaded, and many other concerns need to be taken into account. If you're going to be doing this daily or multiple times a day, it definitely needs to be automated. Fortunately there are a number of tools to assist in this like Ant, MSBuild, or others. Again, it is very worthwhile to have a resource who really knows what they are doing make sure this is set up properly for your team.

- **Automated Test** – You're not going to cover every conceivable permutation or catch every single error, but having automated test to make sure key functions work as expected and things that used to work did not suddenly stop working on the current build is essential. Manual test is time consuming. Automated tests take time to set up but pay dividends every time they are run. Dividends of increased quality and decreased time spent hunting down bugs that have symptoms crop up elsewhere in the system but can be traced back to a root cause that could have been discovered early if the right tests were run to uncover them. Google "automated test tools" and you'll find a great many tools that can help. Better yet, seek the advice of someone who has a lot of hands-on experience with automated test.

- **Developers Committing Code Daily** – If you want to build at least daily, developers should also be working in chunks that can be delivered and built daily. Committing code to be built has the side-effect that compilation problems will be discovered and automated tests can detect errors which can then be fixed very soon after they are created. Since not much has changed since the last commit, there is a much smaller footprint of code that needs to be examined to find the source of problems that have been introduced since the last commit/build.

- **Cloned Production Environment for Test** – There can be all kinds of problems that will crop up when you move between different versions (or have different patches applied) for databases, libraries, operating systems, etc. The last thing you want is to be discovering issues cause by these differences when you move to the actual production environment, so it is essential to have a staging server that is a clone of the production environment used for flushing out any problems before you are ready to promote a build to actual production.

Is this everything? Certainly not! Having a solid infrastructure team in place is the single best investment a company can make to ensure success at continuous integration. This includes people who are experts on software configuration management, build management, automated test, and deployment. Typically an infrastructure team like this can be shared across Agile teams, but care needs to be taken on overloading the members of this team. It needs to be staffed sufficiently to support the number of Agile teams that will be needing their help.

In the end, continuous integration will result in regularly running builds of higher quality, and that reduces the overall risk profile of your software product development. It should greatly reduce the probability of getting blind-sided by time-consuming bugs right before a major delivery or worse yet, bugs that sneak through into deployed software. Bugs should be smaller and found more frequently, which actually results in them being easier to debug and fix. But none of this is free. Reduction in bugs is directly related to the quality, and thoroughness, of the tests you run.

If your code doesn't run at the end of an iteration, it ain't Agile. "Working Software" is a key tenet of the Agile Manifesto: The principle "*Working software is the primary measure of progress"* reinforces that basic thought. The core idea is that the team keeps a working system and minimizes the impact of the inevitable bugs that happen in any software development effort by making small changes, building, testing and integrating. This grows the system daily in a positive way. Embrace Agile and have more working software each and every day with continuous integration!

THE DESARA GROUP

Bob Bretall

Enterprise Agile Coach

Bob Bretall has spent over 30 years in the software development field with hands-on experience in all aspects of the software development lifecycle. He has been a developer, designer, team lead, manager, trainer, mentor and consultant.

Read More **Bob.Bretall@DesaraGroup.com**