



Breaking the Major Release Habit

Can agile development make your team more productive?

Author:
Damon Poole, Chief Technology Officer

AccuRev

Can agile development make your team more productive?

Keeping up with the rapid pace of change can be a daunting task. Just as you finally get your software working with a new technology to meet yesterday's requirements, a newer technology is introduced or a new business trend comes along to upset the apple cart. Whether your new challenge is Web services, SOA (service-oriented architecture), ESB (enterprise service bus), AJAX, Linux, the Sarbanes-Oxley Act, distributed development, outsourcing, or competitive pressure, there is an increasing need for development methodologies that help to shorten the development cycle time, respond to user needs faster, and increase quality all at the same time.

An emerging response to this challenge is a methodology called agile software development, the common theme of which is taking a traditional development process with a single deliverable at the end and splitting it into a series of small iterations, each of which is a microcosm of the full process and each producing working software.

Adopting an agile methodology poses its own set of challenges. It is used mostly by early adopters with small colocated teams, it has little tool support, and though the adoption can be done in phases, getting the full benefits of agile development requires sweeping changes to all phases of the project lifecycle. In addition, agile is actually a large and growing family of methodologies, including extreme programming, Scrum, lean software development, Crystal methodologies, feature-driven development, and many others. Although it is nice to have a wide selection, it does complicate the adoption process. Agile also challenges many fundamental and long-held legacy beliefs about the software development lifecycle. This makes it difficult to achieve the critical mass required to start an agile project.

The Key to Agile: Short Iterations

It has been widely stated that the benefits of agile development are producing value, maturity, and feedback faster; allowing early feedback to inform requirements to produce a better product; and increasing business flexibility. I believe that these benefits are all derived from a single agile development practice, short iterations, and that all other practices are enablers. Therefore, I will focus on short iterations.

The amount of time between releases varies widely from organization to organization and from project to project, but according to a Forrester Research report, "The Expanding Purview of Software Configuration Management" (July 2005), most iterations are still in the range of three to six months or more. No matter what your development process looks like, it is very likely that it can benefit from agile development. In many ways, what you are doing now undoubtedly overlaps with the agile practice of short iterations, whether you realize it or not. If you do one or two major releases per year, you probably don't think of yourself as doing short iterations. But hold on, what about all of those release candidates you produce near the end of a release cycle? Sure, you probably released only a couple of them externally, and even then they were betas, not "real" releases. And what about the three unplanned follow-on releases you did after that major release? How about hot fixes? Customer one-offs? Builds for QA? Demo builds?

If your customer is internal, you probably release once or twice per week or perhaps even per day. Before you start claiming to be agile, however, realize that frequent releases alone do not qualify as agile development.

You may think of all those unplanned and multiple candidate releases as exceptions or as impediments to producing high-quality software. Another way to look at it, however, is that change is an inevitable fact of life and it is better to embrace that change than try to reduce

it. This is exactly what agile is all about, and it includes tools that help to transform what feels like hurtling at high speed out of control with danger at every turn into regularly scheduled coast-to-coast jet flights.

By using short iterations, you are able to get customer feedback sooner. Instead of releasing a major new feature all at one time, if you break it into logical pieces (assuming that this can be done for a particular feature), you can find out sooner which pieces are taking the feature in the right direction and which are not, and adjust your plans as you go.

Let's say that the perfect implementation of a new feature has 12 sub-components. Of course, you can't know in advance what that perfect implementation is. Using a traditional approach, you implement the first version in a year with 12 sub-components. It turns out that six are exactly what are needed, and six are completely wrong (even though that's what was asked for). So, a year later, you release the second version, and now you've got nine out of 12 correct...and so on.

Using short iterations, you could have released a component a month, getting feedback once a month instead of once a year. Using this tighter feedback loop, you should logically converge on the correct solution much faster than you would on a yearly schedule.

Frequent Releases and Quality

The idea of frequent releases tends to be associated with low quality. It is easy to understand this line of reasoning, but it is a misunderstanding of agile development.

If you take a non-agile process and simply crank up the frequency of releases, you are guaranteed to take a hit in quality. Also, if your test phase is three months for a release, the idea of shrinking it down to a week so that you can release every month is pretty scary. But neither of these things is what actually happens. Nobody takes an existing process and simply starts releasing every month, and three months of testing isn't scaled back to just a week of testing.

If you have one release per year and go to a monthly release schedule, what really happens is that you redistribute the many tasks involved in a single release into 12 smaller releases. Each task—writing the spec, doing the design, writing the code, writing the automated tests, etc.—for a particular change still takes the same amount of time. The difference is that instead of occurring sometime during a one-year period, these tasks now occur over a one-month period. The exception is for changes that will take more than a month. In that case you have three choices: temporarily use a large enough iteration to accommodate the change, find a way to break the task up into smaller pieces that will fit into a one-month iteration, or do the change separately and add it to whichever iteration is in progress when the change occurs.

Some tasks, such as doing a full system build or packaging the software for release, are independent of the amount of change in a release. The biggest challenge is reducing the fixed overhead, which generally includes testing, so that the ratio of change to overhead stays at an acceptably high level.

Critical Path Analysis

To do short iterations, everything needs to be adjusted to fit a shorter cycle time. An iteration is really just the time between releases. This is actually easier than it sounds. The key is to determine the critical path of a small development task (for example, a minor bug fix) from initiation to end product and then to determine the amount of time involved. You may be surprised by the result.

Let's consider a slightly different exercise. What is the critical path time for a customer-down hot fix in a high-impact area of the code? Your company probably has a special process for this situation, and once an issue has reached this status, it undoubtedly gets top priority and all of the resources it needs. Thus, there will be no dependencies on other issues, no waiting for somebody to get back from vacation, or any other delays in the critical path other than the time directly associated with the sub-tasks required to produce the hot fix. Since this involves a high-impact area of the code, it should get the highest level of scrutiny that your organization has to offer: multiple code reviews, regression tests, manual testing, the addition of new tests, stress testing, etc.

This type of hot fix typically takes four to eight hours with a worst case of 24 hours. Because time is critical for this issue, testing is limited to a subset of the testing that is done for a regular release. For example, only platform-specific testing or data integrity testing is done first; then the full battery of testing is done post-delivery, which might take another day. Most organizations can turn around a high-quality release that contains just a small change in two days or less if they really have to.

The critical path exercise just described demonstrates that the actual overhead associated with a single small change is at most two days. Since most of that overhead is related either to fully automated testing or to stepping through a manual test plan, and needs to be run through only once, why is it that so many organizations perceive that there is a much larger overhead—sometimes as much as three months for a one-year release cycle—associated with getting a product out the door?

It boils down to two simple answers: breaking the habit of long iterations is difficult, and long iterations hide fundamental problems. To make matters worse, these two problems tend to reinforce each other. The hidden problems help to keep the cycle time long, and the long cycle time helps to keep the problems hidden.

Since everybody “knows” that it takes a long time to qualify a major version of software for release, you of course want to get as much in there as possible. This desire must be balanced against the fact that if you take too long to get the next release out the door, you may fall behind your competitors or miss out on opportunities. As the planned release date comes and goes, you get nervous that it is taking too long and that you're going to break all of those promises you made, so you look for shortcuts. Eventually, the release goes out, but it has some problems and has to be patched a few times before settling down. Everybody remembers that and you vow that next time you won't take any shortcuts, thus reinforcing the belief that shortening the process produces lower quality.

Long Iterations Hide Problems

On the one hand, producing a quality release takes two days of overhead. On the other hand, it takes three months. How can both of these statements be true? This is where the idea of “hidden problems” comes in. The many variations of hidden problems all stem from one root cause: Feedback on process problems comes late in the cycle—too late to take preventive action, so curative action has to be taken instead. Remember the old adage, “An ounce of prevention is worth a pound of cure.”

Let's say that you have 100 planned work items for a release. Of course, the system test comes near the end of the cycle because you know it will be painful and you don't want to go through that process more than once. Therefore, you may do all of the specification and design up front for all of the changes, or you may do it in batches. Perhaps you then move on to doing all of the coding and testing, or maybe you start coding as the batches of specifications and designs are finished.

Regardless of the exact process, you won't start to get information about how good the requirements, specification, design, and coding are until the system test starts. Now you may find that the requirement gathering was done very poorly because the testers can't make heads or tails of how things are supposed to work. If, instead, you had broken the release into 10 iterations, you would have found this out after the first iteration, corrected it, and as a result the impact of the problem would have been reduced by a factor of 10. Instead, you now have to take more time than planned because of the rework required for many or most of the 100 work items, not just in the coding but also in revisiting the requirements, specifications, and design, and rewriting the test plans and automated tests.

The delays associated with process problems being detected late in the cycle occur every release, which is why so much time is reserved for the end game. It isn't the qualification of the release that takes so long—that still takes only two days. It is the process of getting the release to the point that it makes it through that two-day gauntlet, without any problems, that takes so long.

Feature Creep

A year is a long time. During that time, the pressure to add more work items to the release grows and grows. After all, what are one or two more items among 100? Surely there will be time to fit them in during that year!

How soon we forget our arch-nemesis: feature creep. Before you know it, you've been seduced into adding an extra 30 work items to the plan and the code freeze date is upon you. Somehow you have ended up doing all of these small new items but you still haven't done some of the must-have items from the original plan. But that's OK; you still have those three months of testing, so you can get at least some of those items done then. You would like to have a high-quality release as close to on-time as possible. So, painful as it may be, you cut some of those must-have items from the plan and create a new follow-on release right after the main release.

Feature creep seems to be a function of the time between releases. The more features you add, the longer it takes to get the release done, the more opportunity you have for feature creep. A large set of features in a release, compounded by the additional changes required by discovering process problems late, and feature creep all work together to create two additional problems: code churn and increased difficulty of root-cause analysis.

Once system testing begins, there is enormous pressure to fix problems and meet the deadline. Many people are making many changes to interdependent systems. Just as module A, which depends on module B, starts to work with the previous set of changes made to module B, the next set of changes to B comes along and now A no longer works. Plus, the set of changes to A breaks C, and so on. Since so much is changing, it takes a long time to do root-cause analysis to figure out how to solve problems and make fixes. Sometimes it seems that for every problem solved, two new ones appear.

Getting Started with Agile Development

Making changes and adopting new practices is hard. Anything that requires sweeping changes usually encounters a lot of resistance or is done only when the situation is so hopeless that changing it "couldn't possibly make things any worse."

I recommend a gradual adoption of agile development via a series of small process improvements. There is no need to start with a brand new project or to make wholesale changes in an existing project. It doesn't matter what process an organization is using, it can start down the path of agile development today and adopt as much of it as desired at

whatever speed feels comfortable. Two enabling practices that make good starting points are writing tests first and work-item ranking (aka backlog).

Writing tests first. Writing a test requires having a solid understanding of how something is supposed to work so that you can verify that it is working properly and that it fails properly (among other things). Tests and requirements are closely related. If you are unable to write tests, that's a good early warning that you have a problem with the requirements, in which case, it is unlikely that anyone will be able to write the code that the tests are supposed to check. Since writing the tests should take less effort than writing the code, it makes sense to write the tests first as an early warning system for any requirements problems and to reduce the chance of having to rewrite code because of those problems.

Work-item ranking. Work-item ranking, also known as backlog in the Scrum agile methodology, is simply placing the outstanding work items in a list from highest priority to lowest priority from the perspective of your market. This practice can vastly simplify project planning, keep you focused on your target market, and help to prevent feature creep. If a new work item is suggested for the current iteration after work on the iteration has begun, then it is considered for the current iteration only if it ranks higher than a work item that is planned for the current iteration but has not yet been started.

Agile's Position in the Technology Adoption LifeCycle

Although agile development definitely has buzz and growing momentum, not that many people are actually using it today. It still has the feel of being in the “early adopters” or even “innovators” phase of Geoffrey Moore's technology adoption lifecycle. Since there aren't many people using agile development, it will be difficult to find people in your existing network to lean on for help in adopting it. To get the benefits, you will have to sign up for the risk involved in being an early adopter and make a commitment to leading your first agile project to success.

The main reason the software industry exists in the first place is a fundamental belief that people will pay for the productivity gains created by automation. Yet, many proponents of agile development recommend keeping track of project tasks with 3x5 cards on a wall, with their position on the wall indicating their status. Unfortunately, this makes it difficult to take advantage of modern technology to edit, search, categorize, reorganize, report, track, manage, distribute, copy, share, access remotely, and do backups with the project's information. This in turn makes it difficult for team members to work in different offices, at home, or while traveling.

The reason for resorting to primitive tools is in part a reaction to the feeling that software process tools are bureaucratic in nature. There is some truth to that, as evidenced by the fact that most software tools were designed to fit into a framework of traditional development practices and have not yet adapted beyond superficial changes to meet the challenge of agile development.

For example, to adopt the practice of work-item ranking, you will undoubtedly need to resort to using Excel, do some tooling of your own to support it, wait until the feature becomes available in your project management or issue tracking tool of choice, or bite the bullet and migrate to one of the small crop of new agile project management tools such as RallyDev or VersionOne.

Scaling Agile Beyond Small CoLocated Teams

The biggest challenge slowing mainstream adoption of agile development is scaling it beyond small and colocated teams. In many of the agile methodologies, having a small colocated team is either a stated requirement or an underlying assumption. This combined with the deemphasis on automation and limited tooling means that larger and/or distributed teams will need either to experiment with adapting an agile methodology to work in their environment or to wait for others to finish blazing those trails.

Agile Resources

To learn more about agile development, consult the following sources:

The Agile Alliance; <http://www.agilealliance.org>. This is a great place to start any search for more information about agile development.

Extreme Programming Explained, 2nd Edition, by Kent Beck (Addison-Wesley Professional, 2004). This is a seminal book on agile development. Whether you agree with the idea of 3x5 cards or not, this is a short and easy read that is well worth the effort.

The Toyota Way, by Jeffrey K. Liker (McGraw-Hill, 2003). Lean manufacturing, pioneered by the Toyota Production System, is one of the key influencers of agile thinking. In this fascinating book, Liker not only describes Toyota's approach to the lean methodology, but also explains how it is just one aspect of Toyota's total approach to producing high-quality products in an unusually short amount of time.

Lean Software Development, by Mary Poppendieck and Tom Poppendieck (McGraw-Hill, 2003). These folks have done an excellent job of applying the lessons of lean manufacturing to software development. Reading this book was the tipping point for me.

Agile Project Management with Scrum, by Ken Schwaber (Microsoft Press, 2004). As one of the very few "lessons-learned" books on agile, it relies heavily on case studies to illustrate the principles of Scrum as applied in a wide variety of scenarios. This is a good read regardless of which agile methodology appeals to you.

DAMON POOLE is CTO and founder of AccuRev Inc. (<http://www.accurev.com>), an application lifecycle management (ALM) software company that develops software configuration management (SCM) tools focused on improving parallel and distributed development for global teams to remain agile and competitive. Poole has developed SCM best practices over his 15 years managing small and large development projects at companies such as Fidelity Investments and the Open Software Foundation, where he led the SCM tools development effort for the massive multivendor integration projects. Poole earned his B.S. in computer science at the University of Vermont in 1987.