

Agile configuration management for large organizations

Peter Schuh

March 15, 2007

from The Rational Edge: Because of their size and organizational complexity, large companies need to embrace Agile development principles. Read how hundreds, even thousands, can work together efficiently within an Agile configuration management environment.

Early in my consulting career, I was fortunate to be placed on a project that was experimenting with a new approach called eXtreme Programming. The project environment was a rather typical candidate: twenty people, with limited complexity and platform requirements. Overall, the project was a success. The new approach helped us deliver on time and with fewer defects. After that project, I found myself in a variety of situations that were more challenging to an Agile approach, including a large project team (one hundred plus) and fixed cost work. A common theme in all these experiences is that they were all single-project teams, even though some were significantly larger than others. Despite these experiences, I did not fully appreciate the benefit that Agile Development can bring to an organization -- specifically, the Agile practices and techniques related to configuration management -- until I spent almost two years working with several project teams in a prominent Fortune 100 company.

When I arrived as a consultant to this company, I found an entire development organization where there was no routine use of source control, much less any automated builds or automated unit testing. It took several months of hard work just to set one project team on a more Agile course, with continuous integration, short iterations, and various other Agile practices and techniques. When that project was successful, team members helped guide other teams towards an Agile approach. After eighteen months, we had six projects following the major Agile configuration management practices. Each project team had its own codebase, but everyone was sharing components, tests, and build processes. The programmers were checking in code multiple times a day, adding automated tests just as quickly, and recompiling even after writing only a dozen lines
of code. Project teams were running their own automated test suites, as well as those of other systems, multiple times a day. A significant portion of the organization was beginning to benefit from more robust code bases, more timely deliveries, and, ultimately, better end result products.

Since that time, I’ve been approached by many development teams within large organizations wanting to know if Agile is right for them. They've bought into the notion that Agile is disorganized, chaotic, and risky. Nothing could be further from the truth. My own experience has proven that Agile practices and techniques can provide a dependable and flexible configuration management environment that is vital for large organizations to sustain competitiveness and meet their quality objectives.

In this paper, I will present some of the basic building blocks of Agile configuration management, and detail how these practices may be used to benefit large development organizations.

**Before I start: Clarifying terms**

The profession of software development is both very good and very bad about its use of terminology. That is, we all do a great job of coming up with terms. Unfortunately, we typically do a lousy job at agreeing on what they mean. Therefore, before continuing, I want to take a few sentences to clarify the terms I'll use in this discussion.

Starting with the easiest first, a *large development organization* can take a number of different forms. It may, for example, be a large project, numbering a hundred or more individuals, that has been decomposed into a number of subsystems and subteams for planning and development purposes. It may, also, be a development organization that consists of numerous, interconnected systems and project teams. More generically, it is probably any development organization that chooses to describe itself with the term "enterprise." Large development organizations mean multiple teams and multiple codelines. Most often, these large organizations possess a variety of systems in different stages of development, production, or near-retirement; any number of databases, flat-file repositories, and other data sources; a flurry of different project schedules and mandates; and a throng of interested parties with varying needs and agendas. Quite often these large organizations are on the verge of -- or have already reached -- complexity.

At a high level, *Agile Development* is very easy to define. It describes any development approach (typically in the guise of a known methodology employed by an actual project team) that adheres to the values of the Manifesto for Agile Software Development. These values, in brief, focus on individuals and interactions, working software, and customer collaboration, and they acknowledge change as an unavoidable and even valuable component of software development. This high-level definition, however, only describes what Agile teams value, not what they do. When I speak of what Agile teams do, I mean the practices and techniques that Agile teams follow, such as continuous integration, automated unit testing, and short iterations. In the Agile community there are large, ongoing debates over whether a team that follows Agile practices but rejects Agile values can carry the moniker *Agile*. The values are important, because they provide guidance on the appropriate implementation of Agile practices and techniques. However, for the purposes of this paper, I will sidestep this debate and use term *Agile* to identify those practices and techniques that Agile teams use.
And then there is configuration management -- a concept that can be as challenging to describe as "quality" -- which has traditionally held a number of different definitions. Everyone seems to agree that configuration management covers the identification of items within a system and the controlled change of both specific items and the system as a whole. A very narrow definition of configuration management may be satisfied by the implementation and proper use of any popular source control system. Meanwhile, a very loose definition might toss in nearly the entire project team and all its artifacts, including all code and activities meant to ensure the correct operation of any part of the system, all change control activity, and even the tracking of any alterations in the day-to-day procedures of the team. For this paper, I'll take a somewhat middle-of-the-road definition for configuration management, by including any work done by programmers to organize the parts of the system, know the state of the system at any time, manage its evolution, and ensure the continued and proper function of the system throughout the development process.

The need for Agile practices in large companies

Now that we've measured up our ingredients for this discussion, let's see how they mix together. First, while small projects may get away with spotty and informal configuration management practices, most readers will likely agree that a formalized configuration management approach is required for large development organizations. I make this statement, which six years ago might have been considered overly bold, based on my observations of the inherent issues faced by large-scale development efforts. When dozens (if not hundreds) of product components are in play, and you're dealing with hundreds (if not thousands) of developers, the potential for chaos, slow development cycles, and poor product quality is extremely high. Large systems simply become too complex too fast to be sustained with manual systems. In these organizations, automation, process control, change management, and team coordination are necessities to keep development on track.

Second, let's discuss the mix between Agile Development and configuration management. When Agile Development was a new and growing topic of interest among software development professionals looking to break with the old habits of slipped schedules, cost overruns, and failed projects, no one was talking about the Agile approach to configuration management. Agile, however, does have quite a bit to say about what are good configuration management practices because Agile teams need sturdy and flexible codebases in order to be responsive to ever-changing business environments and customer needs. One way Agile teams do this is by requiring that the code is frequently integrated (typically several times a day) across the entire project. Another key principle of the Agile mindset brings testing in as a critical element of effective configuration management. On many Agile teams, all new code is covered by an automated unit test, and all unit tests are run every time a build is performed. A broken unit test is taken as seriously as a compilation error. As in any good configuration management process, Agile teams want to know the health of all their codelines. Furthermore, they work hard to keep the code from ever drifting far from a release-worthy state.

Finally, there is Agile Development and the large development organization. Any large organization really can benefit from the incorporation of some aspects of Agile Development. Granted, there are unique challenges for large development organizations, such as those related to communication and coordination between individuals and teams in addition to the raw logistics activity associated
with multiple projects, systems, and data sources. But these are problems that large organizations will face regardless of whether or not they are taking their cues from an Agile approach.

What does Agile Development have to offer the large organization? First, Agile can increase team efficiency by automating tasks that reduce the likelihood of human error and enable teams to do more work with fewer resources. Second, Agile can help large organizations improve quality and deal more effectively with change by accelerating the speed of feedback loops to development members so problems can be resolved more quickly. Third, Agile can encourage richer and more timely communication by replacing large (and quickly outdated) requirements documents with iterative planning, analysis, and development activities that can be documented automatically by the systems themselves as code is being designed and written.

Finally, it is important to note that an Agile CM approach can be implemented at either the project or organizational level. There is no need to initially turn an organization on its head, since individual projects may be treated as test labs and idea incubators. Meanwhile, when Agile CM practices are implemented at the organizational level, the organization must be careful to allow each project team a sufficient level of flexibility and autonomy to implement the solutions that best fit its individual needs.

### Agile configuration management practices

Streamlined processes and automation are the foundation of an Agile CM approach. Each activity (from checking in code to fixing a broken test) should be easy to perform and provide quick feedback to both the individual programmer and the entire team. Furthermore, Agile teams attempt to make these activities self-documenting. For example, an automated build need only be documented in its execution scripts. One can easily count the benefits of a collection of well-written automated build script over a manual process accompanied by a constantly out-of-date, "how-to" document created in Microsoft Word.

The practices that compose Agile CM have been identified for their usefulness in a wide variety of project environments -- whether large or small, simple or complex. I will discuss the practices themselves in this section, and apply these capabilities to the specific needs of large organizations in the following section.

### Source control

This is the oft-forgotten critical component of Agile configuration management, not because Agile teams do not use source control (they do). It's forgotten because most Agile teams assume that every project has a source control system and that every project uses it correctly. The average source control system comes with a host of goodies, such as versioning, rollback, tagging, and merge assistance. Even more important, however, source control provides a reliable place of record for all the codelines of a project team or development organization. This only happens, however, when every programmer is checking code into the system on a frequent basis. When I say this, I mean at least once a day. When this happens, a project always knows where to find the current system in its entirety. It is not scattered across several development workstations, or possibly a handful of tarballs located somewhere on a shared server. The current system (or something no more than a few hours old) is always what checks out of the source control system.
To reiterate, just because a project or organization has a source control system does not mean that that system will support an Agile CM approach. At one client where I managed several teams, the two hundred person development organization used a proprietary tool for organization-wide source control. But the system had a critical flaw: it took hours to perform a single check-in! Because of this, teams only checked in their code when they had to -- prior to releasing to production. A common source control system can be of great benefit to a large organization, as I explain later, but this is the case only when individual programmers and teams can actually check code in and out in a time-efficient manner.

I'll make one final note on source control. The team should not merely version the code that it is writing; it must also version the process (or scripts) it uses to compile and test that code. This way, if the team ever needs to roll back the code, it will also be able to roll back the build and test processes required to make that code useful and useable.

**Build automation**

An automated build is the first step a team can take toward assessing the stability of their current software or system. Furthermore, an automated build reduces the time programmers spend on unnecessary tasks and removes a bottleneck (namely, the team's reliance on a lone build master or independent build team) from the development process, thereby enabling the team to respond faster to change.

The goal of this practice is to reduce the build process to a quick push-of-a-button activity that any programmer on the team can perform. This activity should include all the code related to the system, regardless of what component or interface a programmer is working on. At the same time, the system must compile quickly. Faster workstations, incremental compilation, and alternative compilers are all strategies that may be used to keep compile times short. For the individual programmer, the ability to quickly build the system while writing new code has a number of benefits. First, it helps programmers verify the correctness of the assumptions made while coding -- for example, to verify that certain external APIs work as expected. Second, routine builds of the code guards against issues that may otherwise silently arise from recent check-ins by other programmers. Finally, it identifies unknown dependencies that may reside in "far off" portions of the system that rarely surface through local builds.

An automated build that is useable by the entire team will reduce the time that team spends chasing down compilation and convergence issues. Programmers no longer need to wait hours or perform a set of arduous tasks to confirm that newly written code compiles. Instead, moments after a programmer has written his code, he will know whether his new code will integrate with everything that has been written before. This means compilation and integration errors will most often become apparent when they are introduced into the system, and can therefore be dealt with quickly and easily.

Finally, automated builds become more important the larger the system gets. Large projects that entail or connect with lots of other systems on a variety of platforms, especially, require some real strategic thinking to put together a build process that makes sense. And obtaining short compile times in these environments can be a challenge. Later in the paper I'll discuss some approaches
to handling builds of larger systems -- that is, when it simply is not practical for a single team to compile and test the entire system on a regular basis.

**Automated migration and deployment**

This is the next logical step following an automated build. The reason for automating migration and deployment activities is to streamline and increase the predictability of promoting builds from development through testing environments and into production. Too often, a myriad of problems arise the first time a project has to coax its way into system testing, and then into user acceptance testing, and then fumble its way into production. With automated migration, teams can regularly perform dry runs that deploy the code into a clean "production class" environment, where automated unit and system-level tests may be executed. By testing in a near-production environment throughout the development process, the team will identify environment, integration, and even performance issues long before they make their way to system testing. This also makes the team much more familiar with the actual process of deploying to production. Finally, human error is largely removed from the equations, when the majority of the production deployment process is automated.

Another important point about migration and deployment is that these activities are typically managed by a dedicated team that may not even report into the development organization. Integrating these processes into the project team's everyday activity creates more effective and timely handoffs between teams. Admittedly, this can be a challenging activity for large enterprises because of their sheer size, reporting structures, and tendency to spread departments across multiple geographies. I'll discuss some more advanced solutions to this problem below.

**Test automation**

Automated tests are the first step a project can take toward knowing whether the current system is release-ready. Automated tests should be written as an accompaniment to all new code in the system. Additionally, when old, untested code is modified, programmers should write new tests for that code. Finally, when a defect is discovered in acceptance testing or production, a test should be written to demonstrate the defect and the new test should be incorporated into the overall test suite to prevent the defect from happening in the future. When all the unit tests in the system pass successfully, a programmer should have a high level of confidence that his code is functioning properly and that he has done no harm to other portions of the system.

From an Agile point-of-view, automated unit-level tests are an extension of the build process. Every time a programmer runs the build process and has a successful code compile, a run of all the applicable unit tests should follow. In an Agile CM environment, a broken unit test is treated with the same seriousness as a broken build. This way, small problems do not fester into big ones. And big problems (such as an entire section of the application that is no longer functional) do not linger silently in the background until one week before the release date. Instead, the team agrees to address such problems as they arise, and the ever-growing suite of unit tests increases the likelihood that errors will be accurately detected a higher percentage of the time.

One thing I'll add about both unit- and system-level tests: Smart projects and organizations will spend some effort on **test data management**. This is the process of making test data easy to...
create, easy to modify, easy to maintain (while the system's data structures evolve), and easy to restore (as in, before every test). A huge suite of automated tests that relies on brittle and arbitrarily devised data structures can quickly be brought to its knees when a major change is made to the data model or even, in some cases, when the development database is wiped clean.

**Continuous integration**

Continuous integration ties together the proper use of source control, an automated build process, and a trustworthy set of automated tests to provide the team with a high level of confidence in both the stability and proper functioning of the system under development. In a continuous integration environment, programmers are writing code, running the build and tests on their own workstations, and checking in multiple times a day. To keep everyone on the team honest, there is typically a stand-alone build machine (or group of machines) that compiles the entire system and runs all the tests in a clean environment that closely resembles the production configuration. This activity can be triggered either manually or automatically -- either at fixed intervals or incrementally as programmers check in code. If, for whatever reason, the code does not pass the build and tests in this clean environment, the team is typically alerted less than an hour after the check-in. In many circumstances, Agile teams will mandate that no one on the team is allowed to check in additional code until the situation is resolved.

The benefit of continuous integration is that it instills a development discipline where individuals are discouraged from checking in poor quality code, and the team is committed to resolving errors immediately when they occur. Since code check-ins occur multiple times a day, programmers in a continuous integration environment rarely or never make changes to the code that will cause them to be more than a few hours from a stable build. Additionally, if there's any checking in of half-baked code, the team's build machines will catch it within hours, not days or weeks. Therefore, programmers have to think through designs, and even test some ideas out, before they cut into the code. This means that programmers in a continuous integration environment are much less likely to perform their experiments directly within the system code, to half-write a piece of functionality and forget to complete it, or take the system apart and leave it all spread out on the garage floor. When errors are detected shortly after the code is written, resolution times are typically much faster than if they were identified days or weeks later. Over time, this practice results in higher quality code and more rapid release cycles.

**Scaling agile CM for large systems**

Large organizations experience many of the same CM problems that small and medium-sized projects do, with a small heap of additional frustrations related to their multi-system, multi-project, multi-initiative nature. For example, while a small project may uncover a defect or experience an integration problem that may take the entire team hours or days to fix, integration issues and defects discovered across systems can cost large organizations weeks of time. Similarly, issues related to source control and versioning on a small project are trivial compared to the collective migraine that results when a system rollback or multiple-application redeployment is performed in a large organization that has no dependable configuration management system in place. Due to sheer scale and complexity, proper configuration management practices are essential for large organizations.
The Agile approach to configuration management can help large organizations address configuration management issues while remaining more flexible to the changing needs of customers, evolving business climates, and ever-advancing technologies. Additionally, an Agile CM approach can help new projects get up and running faster by reusing build processes from existing systems and projects.

In this section, I will discuss three topics related to using Agile practices in the large development organization. First, I'll discuss how an Agile CM process can be implemented flexibly to benefit both the individual project team and the large development organization. Second, how an Agile CM approach can be used on distributed projects in multi-site organizations. Finally, how the organization can increase economies of scale and accelerate software delivery through careful and deliberate integration of applications and processes within their development lifecycle.

Effective team coordination: Sharing codebases and chaining builds

In everyday project environments there is often a need for teams to share code, depend on common libraries, and even share one another's build process. One project, for example, may need to incorporate the build and test activities of other projects. This could include obtaining updated versions of shared libraries to confirm that changes in other projects have not adversely affected the team's code, or validate that changes to the team's codebase will not adversely affect the functioning of another project's code. Furthermore, projects may share essential resources which either project may have the ability to update. Such resources may include common classes or other tools such as test harnesses and test data generators. These situations are common in large organizations, and they often happen without the knowledge or assistance of the organization itself.

An Agile approach provides the opportunity for development teams to collaborate more efficiently and engage in ongoing communication so projects progress more smoothly. By providing rapid feedback on build success/failure, developers are able to detect and resolve problems when they are easiest to fix.

To be effective within a large development organization, Agile practices must be implemented at the individual team level but should be sponsored and supported by a corporate-led configuration management best practices initiative. When implementing an organization-wide Agile CM approach, the individual team must take responsibility for several things. First, it must follow a reliable Agile CM implementation based on the practices discussed in the previous section. Second, it must make its processes available to other teams. Third, and when appropriate, it must include the build process of systems both upstream and downstream into its own build and testing activities. This final step does not need to be done by programmers during their daily activities, but it should be performed by an automated process that runs as often as possible (ideally between once a day and once a week). And, when an issue with another system does arise, this issue must be taken seriously and resolved quickly.

To help implement Agile CM across all teams, the corporate CM organization will also have distinct responsibilities. These tasks may belong to a shared services entity (often called Engineering Services Groups). This organization may provide a common, and user-friendly toolset or platform
which all teams can employ to complete and share their source control, build, and testing activities. This platform may include components such as source control, build, and testing systems. Furthermore, the organization should provide support and guidance for teams in the implementation and continued use of Agile CM practices, and may provide a set of reusable CM processes or best practice recommendations to bring consistency and reliability across project teams. Finally, the organization must ensure that every team still has sufficient control of its own CM process. This may seem like a bit of a balancing act, but it is necessary for effective software delivery. Ultimately, it is still the individual team that is building the software and it is the job of the organization to help each project maximize its success.

Supporting distributed teams and organizations

It is true that many Agile methodologies were not created with distributed teams in mind, but this common characteristic of large organizations simply cannot be ignored by any movement that aspires to alter the software development industry. Despite a lack of specific focus, an Agile CM approach can be remarkably useful to projects and organizations working within a distributed team environment.

To benefit from an Agile CM approach, distributed projects and organizations must leverage the solid implementation of several Agile CM practices, especially routine use of source control, continuous integration, and automated testing. The importance of frequent check-ins and the maintenance of a stable build cannot be overstated, because teams split across time zones and continents need to be confident that they will have access to a complete and operational version of the system on a daily basis. When something is broken or appears out of date, there will often be no one available at a team's sister site to lend a hand.

In order to maintain an Agile CM solution for a distributed project, everything must be checked into source control, including build scripts and local environment settings. Any change that occurs at one site should automatically replicate over to the other development sites. This is needed because of the complexities associated with distributed teams and their typically large systems. Specifically, once a system in development begins to exhibit quirky behavior at one site and not another, days can be lost at both sites before the source of the issue is discovered to be some setting on the server or virtual machine that no one would have ever thought capable of causing such trouble.

Additionally, everything related to the database needs to be replicated and shared. This may be accomplished by scripting all changes to the database and checking them into source control. It might also be accomplished through some form of database replication. Finally, the project must account for any connected or third-party systems against which development activity must be performed. In this case, each site must have access to either the same or identical systems.

There are two general approaches that may be taken to implement a distributed and Agile configuration management environment. The first is built around a single development environment that is constantly accessible by all development teams. This environment would include -- at least -- a single source control system, all databases and connected systems, and the ability to perform continuous integrations. This solution can work very well for teams that work
within nearby time zones and have dependable online access. The second approach is built on the concept of stand-alone development sites. Here, each team has a completely independent and identical development environment, including source control, databases, additional system installations, and continuous integration setup. A daily replication schedule must be put in place and adhered to in order to keep code, data, and environment changes across all development sites in sync. As much as possible, synchronization activity should be automated. Furthermore, automated tests must be routinely written and executed. If daily replication and thorough testing are not performed (that is, if things are allowed to fall out of sync) the organization may find itself with a convergence nightmare on its hands. Finally, projects and organizations can also pursue a middle-of-the-road solution, where some portion of the Agile CM environment is centralized while the rest is maintained at the individual sites. For example, an organization may have common source control and build systems, but maintain local instances of the database and other third-party systems across its different development sites.

**Scalability through flexible integration of tools and process**

If sufficient thought and preparation is put into the creation of good build processes and automation it can become a powerful development asset, and this infrastructure can (and should) be leveraged across multiple projects. A typical inefficiency that occurs in large companies stems from each project team creating a new build system for each software project. The result is multiple custom build applications to maintain with dedicated hardware resources and configuration management staff. This prevents large organizations from gaining economies of scale from the pooling of resources, staff, and best practices knowledge.

If an organization plans to implement Agile practices with any scale (meaning simultaneous code-build-test-deploy cycles across multiple teams, projects, and/or operating platforms), then serious thought should go into how these systems will communicate and interact to create a smooth code-build-test-deploy cycle. If cross-team, cross-system integration is not factored into the overall development strategy, teams often find that development progress is slowed by gaps, wait periods, and miscommunication between the functional silos. Without an infrastructure that tracks and aggregates information from each phase of the cycle, teams can find it difficult to determine the true health and state of a release.

Integration should include workflow automation as well as information sharing (or at least extraction) from your core development systems mentioned in the previous section. Workflow automation includes the order and orchestration of tasks, and should involve a rules-based capability to alter task execution and notification based on the success or failure of preceding steps. When determining your integration approach, teams should look to industry standard approaches (XML, etc.) in order to architect a flexible solution that can adapt to changing needs and development applications.

To find out how IBM Rational Build Forge may be used to rapidly enable an Agile approach in a large development organization, [click here](#).
Notes

1 The Agile manifesto can be found at http://www.Agilemanifesto.org

2 For an informative and sometimes amusing read, see the collection of definitions for configuration management begun by Bard Appleton on the CM Crossroads Wiki, at: http://www.cmcrossroads.com

3 Portions of this section have been adapted from my book, Integrating Agile Development in the Real World, where these practices all are described in much greater detail.

4 For more information on setting up and managing a database in this fashion, consult my paper "Agility and the Database."

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)