Crossroads News
A Monthly Publication for
Software and CM Professionals

Agile SCM – Build Management for an Agile Team

Steve Berczuk, Brad Appleton, and Steve Konieczka– October 2003

A number of people work together to develop a software application. The application is useful only when the components each person works on come together – the process of integration. The mechanics of integration happens during a build. Last month we discussed continuous integration. “Integration” implies “building” and this month we’ll talk about the various kinds of builds one does during a development process.

Some Definitions

To make sure that we understand each other, let’s start by defining some terms that we’ll be using in the article. Hopefully the definitions are similar to the ones that you use; but if not, there should be a simple mapping from your terms to ours.

A *Build* is the process step where you integrate all or part of your application. The build can include steps like retrieving the correct components from various locations in the *Repository*, compilation, running scripts that generate source files from meta-data, etc. In short, anything necessary to go from a set of source files in your version management system to a component that provides value to someone. The product of a build can be a complete application or a component for distribution.

The inputs to a build are “*source components*” which include source files, meta-data files, and pre-compiled third-party library components.

Builds can be full (a.k.a. "clean") or incremental (a.k.a. "dirty"). An *incremental build* makes use of a dependency checking system so that you only build derived components from source components that have changed since the last build. A *full build* starts with no derived components.

Builds are often orchestrated by build scripts, which can be makefiles, Ant scripts, batch files, etc.

Building and Agility

Agile software development is based on a feedback loop between the developer and the customer. The development team implements functionality based on customer input, the customer evaluates the current state of the product against their list of requirements and the schedule constraints, and the team starts a new iteration to implement the next set of functionality.

This feedback cycle works best when the time between iterations is “small.” What “small” means depends on your team and process that you are using. XP teams typically use iterations of 2-3 weeks, Scrum teams use slightly longer cycles. Regardless of the customer iteration length, it is important to monitor the state of the process so that the system does not degrade unexpectedly. On the other hand, if you spend too much time on monitoring, you will have that much less time to do development of new features.

Important as it is, you should develop build (and deployment) processes early on in the project so that they evolve as the product evolves and so that they don’t take up much time. The build is an infrastructure step that helps you to create a product. It is not an end in itself.

Agile software development depends upon a hierarchy of build practices that allow you to balance the need for speed and the need for stability.

Writing build scripts early does some of the same good things for your architecture as test-first development does.

Kinds of Builds

We've identified three patterns for builds: *Private System Build*, *Integration Build*, and *Release Build*. Each type of build has a different scope and frequency to resolve a slightly different set of concerns for its target consumer.

- **Private System Build**, for building a system to use for testing in your Private Workspace. The consumer for this build is the developer doing ongoing work.
- **Integration Build**, to integrate everyone's changes in a central place. The primary consumer for this build is the development team, both for the purpose of having additional assurance that the code integrates in a "clean" environment, as well as for integration level testing in advance of the release of the software.
- **Release Build**, which packages the software for release. The consumers of a release build are the testing team (for pre-release) testing, and the customers of the development team (for production use).

Private System Build

The *Private System Build* is used by the individual developer working on an isolated development task to verify the consistency and correctness of their changes before committing them to the codeline where they will be visible to the rest of the development team. The developer creates a Private Workspace and populates it with the latest working versions from the codeline in order to begin work on their development task. After making changes in their Private Workspace, the developer needs to ensure that the source changes work correctly and won't "break the build" for the rest of the team. The usual sequence of steps for doing this is part of a Private System Build:

- Perform a *Workspace Update* to make sure the changes in your workspace incorporate the most recent changes that have been successfully integrated to the codeline from other developers. This may cause some files to be populated with new versions in the workspace. Other files may have parallel changes and will need to be merged to reconcile any potential conflicts.
- Perform a build (usually an incremental build, but sometimes a full build) and run tests to ensure the resulting components build and execute correctly.

The above might happen multiple times during a development task, at progressive "checkpoints" of development functionality for a requested feature, fix or enhancement. Or it might happen once at the very end of a (small to moderate) development task, prior to committing one's changes to the codeline. In either case, it should always happen prior to committing changes to the codeline. Some kind of synchronization mechanism is often used to serialize commits so that only one person commits their changes at a time (that is another set of patterns for another article).

So the organizational scope (visibility) of the private system build is the individual developer, and the frequency is one or more times per development task. The purpose is to insulate the development team from any problems that might impede the progress of other team members and active development tasks. By ensuring the changes for a single task are correct and consistent with the latest stable state of the codeline, the impact to the team and the codeline is minimized. The team is able to be more "agile" and to integrate their changes more smoothly in a regular and frequent fashion.

Integration Build

The *Integration Build* is used by the integrating developer (or integrator) to incorporate the most recently completed development tasks into a single, stable, consistent and correct result. The result of a successful

integration build is also what is used during a Workspace Update to synchronize the state of an in-progress development task with the latest stable state of the codeline.

An integration build is preferably a full build (if time and bandwidth allows) whose results visibly impact the progress and coordination of the entire development team and all the active development tasks at that time. The frequency of an integration build may be as often as one per development task (at the end of each task) or may be every few tasks (or even daily/nightly). In some cases, there may be two "flavors" of integration builds: one that is done after every development task and does only an incremental build (and possibly runs only a subset of tests), and one that runs daily or nightly and does a full build and runs a more complete set of tests. In either case, the purpose of the integration build is to centrally coordinate and synchronize all ongoing work in small bits and pieces to avoid "big bang integration" at the end of a release/iteration, and to provide a regular rhythm for the development team to tackle integration issues early and aggressively, one small piece at a time.

Release Build

The Release Build is the "formal" releasing process for the application. Typical steps of a Release Build could include:

- Apply a label to the codeline,
- Get a copy of the codeline (workspace or instance) on a "golden" build machine,
- Build the application,
- Possibly test the application and report on those tests,
- Package the targets of the build by either checking them into the version control tool and label as appropriate, zip up the release for distribution, or possibly even run installer software against the release to package it for distribution,
- Email a report showing the results of the build.

While the integration and release builds are similar in many ways, the primary difference is that the release build is intended to package and store the build results for installation and distribution purposes. The integration build is intended to test the quality of the codeline and offer feedback to the development team while the recipient of the Release Build is typically the developers' customer.

To be even more specific about the differences between an Integration Build and a Release Build, the following points are worth noting:

- You typically don't apply a label with an Integration Build, you do with a Release Build
- Integration Builds typically operate on the latest (tips) on the branch, not always the case with a Release Build
- Integration Builds run exhaustive tests on the build, not always the case on a Release Build
- Release Build packages the release for installation and distribution, mostly not the case with Integration Builds
- Release Builds include automated CM reports like BOM (bill-of-materials) and CR (change-request) reports - rarely the case with Integration Builds

The last point worth noting with Release Builds is the fact that they occur only when the development team intends to package a release of software for distribution (Test or production environments). Therefore, Release Builds occur far less often than the typical Integration Build.

Scale and Agility

We have identified the following types of builds and noted that each takes place at different levels of scale. One of the varying levels of scale is that of visibility (or organizational scope/impact). Another varying level of scale is that of frequency and duration (or project planning scope). At each level of scale, a slightly different set of needs and concerns must be resolved using a slightly different set of trade-offs:

Build *(What)*	**Visibility** *(Who)*	**Frequency** *(When)*	**Purpose** *(Why)*
Private System Build	Individual Developer	One or more times per change task	Provide feedback to the developer of the changes just implemented in the private workspace.
Integration Build	Whole Development Team	After each change task and/or one or more times per day (e.g. Nightly)	Integrates the latest changes in the repository and runs automated tests against the release providing feedback to the development team.
Release Build	Independent Test (QA or V&V) Team and Customers	At the end of each iteration/release	Packages release for distribution to the development team's customer.

It's useful to note that while the various kinds of builds work with the same inputs (code, tools, build scripts, etc) and should be similar in many ways, they differ in subtle, but important details. Allowing for these small differences helps you to be more agile and focus your energies on the important task of the moment. Each build should be as close to the final product, but no closer than necessary. For example, the Private System build may suffer from subtle errors because of a slight difference in the developer's toolset, or because the developer chose to do an incremental rather than a full build. But if the choice is between doing an incremental build (at a price of a small risk of failure) and only making one change a day (because the build takes too long), some organizations might choose speed.

Since each type of build requires a greater level of discipline and precision, it can catch anything that slips through in the earlier steps. The integration build can start with a clean workspace and be a full build that is done using "approved" configurations. Finally the release build will add finishing details that are not likely to cause errors, but which are time-consuming. This is not "sloppiness" or "laziness" but a realization that there is always a tradeoff between speed and stability.

These levels of scale are similar to the relationship between pre-checkin testing. A Smoke test may be adequate for many purposes even though it may not catch every problem that a long-running regression test does.

Having said all of that, the levels of scale are not a reason to defer having a process for all of your steps at the beginning of a cycle. Do not wait until the end of the product cycle to do a Release Build, or to address other deployment issues.

Of course, building the application is only part of the story. You need to do appropriate tests (Unit, smoke and regression) to achieve a good system.

Continuous Building

The various types of builds are related to the idea of "Continuous Integration." By taking the build and deployment process seriously throughout the development process you are helping to ensure that the

product makes steady steps forward. If one of the builds breaks, you at least have an early warning of problems in the codeline. Building early and often also helps to ensure that you don't waste time on mechanical processes. Building early and often means that you can ensure build reproducibility (the build scripts work consistently, so you know how long a build will take, and the results of the build are useful for testing).

Building often (at all levels) helps you to ensure the integrity of your codelines. And the state of the codeline is a key to facilitating communication among stakeholders.

Building Agility

The value in leveraging these build patterns is that they are easy to do and repeatable. Where possible, use tools that fit into the development environment, and limit manual procedures. Even the best intentioned, most detail-oriented developers are susceptible to skipping a step if the processes are not automated. The agile community also values documented processes – not "double documented" processes, but documented processes. If you have an automated build process that produces an installable release, you have documented the process. Any developer/build manager can pick up the build script and determine the steps to building the application. "Double documented" processes are manual processes that contain written documentation, that oftentimes is out of date. In the case of having a process with outdated written documentation, it's difficult if not impossible for a developer or build manager to execute the build process without having some other knowledge of the application before starting.

There are some tools and issues to consider when implementing these build patterns:

> *Build Dependencies*: It's very important to manage build dependencies by keeping them up-to-date (e.g. in Make or in ANT), and minimizing them for better recompile-times and fewer "conflicts." Keeping these dependencies clean and up-to-date pays back time and time again, providing many of the same benefits of keeping automated test scripts current. A common practice is to use utilities and compiler-flags to automate the process of updating build dependencies (often making them another form of build-target, e.g., "depend").
>
> *Build Architecture*: The Build system "architecture" (e.g. what aspects of the directory tree structure are the way they are, and how things get split up across multiple Make/ANT files) should support reproducibility, testability of smaller units, and faster builds (and/or parallelism). The structure of the source directory tree needs to meet several sets of needs. Sometimes build-time performance dominates certain source-tree structuring decisions about the location of frequently accessed interfaces and definitions (e.g., "header files"). Symbolic links or "short cuts" can help minimize the impact upon the perceived "closeness" of related files that are often modified together.
>
> *Multiple Build Types, Single Build Mechanism*: Even though there may be several different types of build, it is usually easiest to employ the same or similar mechanism to initiate each kind of build. The same set of files and scripts can be used to define (and parameterize) the different types of build and a simple change in the invocation or environment can specify which kind of build is to be produced using the same basic build scripts and infrastructure.

Improving Build Performance

The "agile" slant on this column is really about trying to accomplish SCM standards while actually improving developer productivity and software quality. With this in mind, one of the key benefits of automating a build is to create a baseline from which to improve build performance. Too often, build times take too long - sometimes more than one hour. Developers get around this in sometimes unhealthy ways to decrease the impact on "their" development times. Through automation, there are some of the ways to improve build times on projects that are large and/or have long build times.

- Storing intermediate targets in the repository to improve incremental build times;
- Appropriately breaking projects into independently buildable modules that create run-time dependencies (project architecture);
- Throw more hardware at it - we can again show how 10 minutes saved on a build process can save, for a single developer doing 6 builds per day, one hour of productive development time each day. It doesn't take many days to pay for a faster build machine;
- Different build utilities can produce very different response times. Consider compile and link times when considering your compiler and linker for your project. There are often several alternatives.

Building Values

Earlier we discussed some values for Agile SCM. Here is a brief review of how using good build patterns fits in with the "agile" values of the Agile Manifesto:

> *Individuals and Interactions over Processes and Tools:* SCM processes and tools should support the way that you work, not the other way around. Pick tools that fit into the environment so that developers will use them. There are many tools that have good qualities that do not help the quality of your codeline because no one uses them. A favorite example: tools that are so expensive that you can only have a few people doing builds. Everyone on the development team should be able to build (at least a Private System Build) at will.
>
> *Working Software over Comprehensive Documentation:* SCM can automate development policies and processes with executable knowledge rather than have the team rely on documented knowledge. Describe your build processes using scripts rather than procedures. This is especially important if you find yourself working late.
>
> *Customer Collaboration over Contract Negotiation:* SCM can facilitate communication and interaction among stakeholders and help manage expectations. Provide the builds that your customer wants/needs. Build frequently so that you can manage expectations.
>
> *Responding to Change over Following a Plan:* SCM is about facilitating change, not preventing it. By building frequently and appropriately you and your customers will have a better idea of how things stand and what to change.

Building sounds like an obvious process, since you must build to have running software. But it is also an item where small improvements can make a big difference in your productivity as a team and in the quality of your software. As SCM Professionals, we appreciate this more than anyone.

Brad Appleton is co-author of H**Software Configuration Management Patterns: Effective Teamwork, Practical Integration**H. He has been a software developer since 1987 and has extensive experience using, developing, and supporting SCM environments for teams of all shapes and sizes. In addition to SCM, Brad is well versed in agile development, and cofounded the Chicago Agile Development and Chicago Patterns Groups. He holds an M.S. in Software Engineering and a B.S. in Computer Science and Mathematics. You can reach Brad by email at H*brad@bradapp.net*H

Steve Berczuk has been developing object-oriented software applications since 1989, often as part of geographically distributed teams. In addition to developing software he helps teams use Software Configuration Management effectively in their development process. Steve is co-author of the book H**Software Configuration Management Patterns: Effective Teamwork, Practical Integration**H. He has an M.S. in Operations Research from Stanford University and an S.B. in Electrical Engineering from MIT. You can contact him at H*steve@berczuk.com*H His web site is Hwww.berczuk.comH

Steve Konieczka is President and Chief Operating Officer of SCM Labs, a leading Software Configuration Management solutions provider. An IT consultant for 14 years, Steve understands the challenges IT organizations face in change management. He has helped shape companies' methodologies for creating and implementing effective SCM solutions for local and national clients. Steve is a member of Young Entrepreneurs Organization and serves on the board of the Association for Configuration and Data Management (ACDM). He holds a Bachelor of Science in Computer Information Systems from Colorado State University. You can reach Steve at H*steve@scmlabs.com*H