Crossroads News

A Monthly Publication for Software and CM Professionals

Continuous Integration – Just another buzz word?

Brad Appleton, Steve Konieczka, Steve Berczuk – September 2003

Last month we wrote that we would be addressing some questions and concerns raised by readers who gave us feedback on previous articles. We still intend to address these concerns. However, since the theme for this month (Continuous Integration) is one of the core "enabling practices" of agile methods like Extreme Programming, we felt it necessary to shift our focus this month to cover it instead of what we had originally intended.

Therefore, this month we will talk about Continuous Integration – is it just another buzz word? We will discuss in detail what Continuous Integration is, what's required to implement continuous integration, and what value it brings to the project.

What is Continuous Integration?

One of the most common activities for SCM practitioners is the software build process. The objective is to create reliable releases of the software in a predictable way. The Continuous Integration approach was created to assist in this endeavor.

Continuous Integration holds the developer responsible for their code changes integrating with other developer's code in the repository before checking it in. Second, Continuous Integration recommends periodically executing a build against the latest source code on a clean build machine for the sole purpose of running tests against the software to identify integration problems as early in the process as possible. The results of these builds are for developer use only as a feedback loop to some quality measure of the latest source code in the SCM repository. If the tests do not pass, the developer is responsible to either back out their change, or to fix the problem and re-integrate.

Continuous Integration primarily effects the software development process, but must be supported by the SCM components of *version management* providing an easy way to update the private workspace, and an *integration build script* which controls the integration build. The more often integration builds are scheduled, the better – several times per day is typical. Developers that check in and integrate several times per day have less change to integrate than developers who wait for weeks before updating their workspaces and/or check in their changes. When a developer is ready to check in a changed file to the SCM repository, the first step is to always update her workspace with the latest code from the repository. She then will perform a private workspace build, and finish by executing adequate smoke tests and/or integration tests. If the tests pass, then and only then may she check in her code.

The second part of Continuous Integration is the automated Integration Build. Again supported by SCM, the automated integration build runs periodically, say every hour, on a sanitized build machine. The build starts by getting the latest code from the repository, builds it in its build area, then runs a series of automated smoke and/or integration tests against the release. The results are then made available via an intranet web site and/or email.

For Continuous Integration to be effective, it is very dependent on two main factors: A completely automated build, and some form of automated tests.

The Automated Build

A fully automated build process can bring many benefits to a project including more predictable releases, shorter build times, documented and versioned process for building the application through the build script itself, and the ability to schedule a build without manual intervention. As stated earlier, Continuous Integration depends on an effective automated build system being put in place.

An effective automated build is one that addresses build-time dependencies when figuring out what objects to compile. This is especially important when using Continuous Integration on larger projects because this type of build system will only build the objects that are affected by the objects that have changed since the last build. Tools such as Openmake, Jam, Ant, and make all handle these dependency conditions quite well.

Once you have your build fully automated, it can be wrapped with scripts to implement a consistent private workspace build, fully automated integration builds, and a completely automated formal release build.

Automated Tests

The second component that Continuous Integration depends on is automated tests. Many people believe there is no way you can create enough automated tests to make a difference in the quality of the software. Even a small amount of tests can make a difference.

Automated tests can be broken down into several categories: Unit tests, smoke tests, and functional tests. Unit tests are tests typically written by developers to test individual methods, classes and/or groups of classes. Extreme Programming implements a test first approach where the developer must write an automated unit test prior to writing a line of code, then write the code to satisfy the test. Unit tests are excellent for Continuous Integration, however, there are few projects that have already established a test harness full of automated unit tests. For methodologies that are not so test-driven, focus time on automating smoke tests and functional tests.

Smoke tests are very high level tests that exercise the most basic of functionality in the application to determine application quality but with very low confidence. For example, you can start the application, log in, view a record, and close the application gracefully. Even though smoke tests may not verify quality with much confidence, this is usually an excellent place to start if your project has no tests to begin with. Take these smoke tests and integrate them into your integration build process. You may question the value of running smoke tests routinely on the latest code, but it is not uncommon for projects to go for weeks or even months without being able to create a build that can pass even a smoke test. Many times these projects don’t know how far they are from having working software.

Once the smoke tests are incorporated into the integration build process, begin adding automated functional tests. When choosing what features to automate, prioritize them based on importance to the application, feature stability, and integration with other features. The highest priority features are those that are important to the application relatively stable and contain interdependencies. Having interdependencies with other features means that changes in somewhat unrelated features could affect the behavior in this feature. An automated test for this particular feature could reap benefits down the road.

The investment involved in creating these automated tests may seem high, but if you hook them up to an integration build that runs 3 times per day, each test will be run against the software at least 60 times per month. Every time they run they’re helping to catch integration problems early, when they’re easier to fix.

SCM Benefits of Continuous Integration

So the million dollar question is, “Is all this worth it?” Are your formal builds today non-events? Would it be worth the up-front costs to eliminate the late Friday afternoon builds that go well into the weekend because of integration problems? In the end, Continuous Integration produces higher quality software by offering developers appropriate feedback at a time when they can use it to improve the software. Along this same line, it’s not uncommon to see a team go through a tedious 3-day release build process and installation into the test environment only to find out that the software release doesn’t pass even the most basic smoke test. Continuous Integration catches these problems early and makes release builds much more reliable and predictable.

We can all agree that when implementing changes to software, integration issues must be addressed at some point. Addressing software integration issues at the point when the integration is developed is far less expensive and produces higher quality results than waiting until the formal release is created and fails.

Continuous Integration helps to establish a “team” mindset for the build, whereby if the build breaks, it is a team-wide priority to get it fixed. Traditionally, the release build oftentimes integrates code changes for the first time, causing build problems that are left up to the build engineer to deal with. Many times the build engineer ends up removing features from the build to get it to work. By incorporating these integration activities earlier in the process, more features end up in the build and the effort typically ends up being more of a team effort.

Recap

Incorporating Continuous Integration into your project may seem like a daunting task, but it’s not all that bad. You likely already have a fully automated build environment, so the biggest task will likely be incorporating automated tests. Don’t be afraid to put the integration build in place with no tests to start with and add them as you go. Just verifying periodically that the latest code in your repository actually compiles adds value. Most good things are implemented and improved upon incrementally, so start small and build onto it. Your developers will become more productive. Your software will be of higher quality and release builds will become non-events. It won’t be long before you wonder how you survived without it!

SCM Specific Concerns with Continuous Integration

Who owns the build?

Now that we’re recommending adding a new build type to the development process, the question is, “who owns the build?” As discussed earlier in this article, this build is owned by both the developers and the SCM group. However, it’s also important for this build to be an exact replica of the formal release build that the SCM group is responsible for, so we’ve found that a team effort between developers and the SCM group is very important. The SCM group is there to provide support for the automated build process and the build is initiated as well as the results interrogated by the developers. Development takes responsibility for the build and SCM takes responsibility for the build process, and the two must work together as a team.

How should we track Integration Builds?

Integration builds are not intended to be tracked beyond the latest “successful” build results. Therefore, no label is applied to the SCM repository to track integration builds. The only important item to store for these builds is the last successful applicable targets, in case the development group should want to install and test the latest. This implies that the prior successful build results be temporarily stored off during the integration build so that they’re not overwritten, just in case the latest integration build turns out to be unsuccessful.

How should we handle multiple concurrent releases?

The Continuous Integration process is set up for each codeline. Some codelines will choose not to leverage Continuous Integration, and others will. You may have multiple concurrent active development codelines, each having their own Continuous Integration build process implemented. There are several situations that can occur when doing parallel development where we need to propagate a change from one codeline to another. This should be done in a workspace first and the developer should verify the integration of that code prior to committing it to the SCM repository. The receiving codeline’s Continuous Integration process will ensure proper integration. Code integration is not a magic event. It involves bringing changes together from two areas that were unaware of each other and testing to see that they play nice together. Therefore, in environments that implement Continuous Integration, it is not recommended to merge codelines without the intermediate step of merging them into a workspace and performing integration testing prior to checking the changes into the repository.

Can Continuous Integration and Component Based Development work together?

It appears on the surface that these two approaches conflict with one another. However, there may be some room for a hybrid solution.

Component Based development teams break off into their own areas to develop components without integration with the other component development teams. Creating unique active development codelines for each component is oftentimes the way to manage this. If you can make the assumption that the components will never intersect, then you can have them all part of the same codeline. The thought is that there are benefits to waiting until the latest possible moment to decide which component versions make it into the release. This approach typically involves a significant integration effort just prior to the formal release build, and almost always creates integration problems the first time all of the components are brought together.

The hybrid approach starts with implementing Continuous Integration within the component teams to ensure that the component codeline is of high quality and integrated within itself. Second, always integrate a component codeline to the release codeline or release workspace, independent of a formal release build. This process can even be automated – upon completion of the integration building of the components, the successful component builds are brought together in an “assembly integration” and form a “system-level integration build.” As a final thought on this topic, there should never be a codebase convergence (branch merge) that doesn’t involve a workspace integration effort with developers and the SCM group working together in harmony to ensure proper integration before checking in the integrated workspace changes.

Up Next!

Next month we will address in more detail the different build types, workspace management and coordination for incorporating changes into the SCM repository.

References

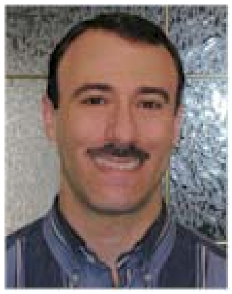
[1] *Continuous Integration*, by Martin Fowler & Matthew Foemmel; see www.martinfowler.com/articles/continuousIntegration.html

[2] *Integrate Often*, by Don Wells; see www.extremeprogramming.org/rules/integrateoften.html

[3] *Managing Complexity: Keeping a Large Java Project on Track*, by Tom Copeland; see www.onjava.com/pub/a/onjava/2003/09/10/dashboard.html

[4] *Characteristics of the Agile SCM Solution*, by Brad Appleton, Steve Berczuk, & Steve Konieczka; CM Crossroads Newsletter, June 2003

Brad Appleton is co-author of **Software Configuration Management Patterns: Effective Teamwork, Practical Integration**. 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 *brad@bradapp.net*



Steve Berczuk is an Independent consultant who 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 **Software Configuration Management Patterns: Effective Teamwork, Practical Integration**. 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 *steve@berczuk.com*. His web site is www.berczuk.com

Steve Konieczka is President and Chief Operating Officer of SCM Labs, a leading Software Configuration Management solutions provider and maker of Quartet™. 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 *steve@scmlabs.com*