Software Configuration Management Best Practices for Continuous Integration

As Agile software development methodologies become more common and mature, proven best practices in all phases of the software development life cycle are of increasing importance. Without such practices, attempts to improve software quality through Agile methodologies can fail, denying software development organizations an important opportunity to increase the business value delivered to customers. This paper introduces the concept of continuous integration and outlines several proven software configuration management (SCM) best practices to consider when implementing a continuous integration environment.

Overview of Continuous Integration

Continuous integration, one of the foundational aspects of Agile software development methodologies, is defined by Martin Fowler to be "a fully automated and reproducible build, including testing, that runs many times a day. This allows each developer to integrate daily, thus reducing integration problems."[1] By extending the idea of a nightly build, where code changes are built and tested nightly, continuous integration helps reduce integration problems and identify and resolve problems more quickly.

With continuous integration, developers are encouraged to update the shared source code repository frequently, ideally several times per day. After each update, there is an automated build and test cycle. The results of the build and testing are reported to the entire team. This tightens team-wide communication about the quality of the committed changes. Since developers are aware of problems earlier and there are fewer changes to look through when debugging an issue, there are typically fewer bugs during the development process and in the production code.

Software development organizations can incorporate software configuration management best practices when designing and implementing a continuous integration system to further reap its benefits. These best practices include the following:

- Using an SCM system to store and version all source code
- Utilizing private developer workspaces
- Enabling local developer builds
- Frequently updating code in the SCM system
- Establishing a staging and isolation hierarchy
- Automating builds at all stages in the hierarchy

The remainder of this paper describes these practices and offers practical advice on how to best configure and use an SCM system to optimize a continuous integration environment.

[1] See Martin Fowler's continuous integration web site: http://www.martinfowler.com/articles/continuousIntegration.html.

Best Practices for Continuous Integration

Use an SCM system to store and version all source code
Software development commonly involves parallel development, with multiple developers making changes that need to be integrated in order to build a product. Further, changes made in any part of the system need to be tracked in order to identify and fix any potential problems. For this reason, it is important to employ a software configuration management (SCM) system to strictly version changes to the code base. In addition to versioning source code, everything needed to build the system should be placed under version control, including the following:

- Third-party libraries
- Properties files
- Database schema
- Test scripts
- Install scripts

All developers should have at least read-only access to all files needed for the build and should obtain all such files directly from the SCM system. This approach ensures that developers are working with the latest build environment, and is preferable to the common but error-prone practice of placing such files on a shared file server.

To the extent possible, developers should obtain the latest source code and configuration files from the SCM system in an automated fashion, to ensure that the correct configurations are always being used. This is especially important in the case of geographically distributed development, where keeping changes in sync between groups in different physical locations and in different time zones can be challenging when using a shared file server approach. To effectively implement continuous integration, all development groups should work from the same central source code repository so that the latest changes from other developers are easily and immediately available.

Utilize private developer workspaces
In order to fully realize the benefits of continuous integration, software development organizations need to ensure that developers can remain productive regardless of the overall state and stability of the project source code. To achieve this, private workspaces that give developers full SCM capability should be used. Private workspaces enable developers to

- work in isolation;
- revert to known “good” states when needed;
- checkpoint their changes; and
- share only mature, well-tested code with other team members.

The benefits of isolation are bidirectional—it protects developers from incoming changes, and protects the shared code configuration from incomplete or incorrect changes from any one developer. By creating private workspaces, developers receive all the benefits of SCM for their personal use, including the ability to revert to a previous state, viewing and tracking of changes between software configurations, and setting aside changes to begin work on a different task.

Once a new known good state is reached (for example, when a developer completes engineering and testing work on a feature), developers should checkpoint their work, typically by “checking in” or “keeping” the local changes in the SCM system. Different vendors use varying terms for this activity, but the basic idea is the same: to save the work at the SCM server in such a way as to make the changes persistent and easy to retrieve. The checkpoint ensures that the developer’s work is safe on the SCM server and that the checkpoint can be revisited at any time. However, since the changes have not been shared, other developers and teams are not affected.

When a developer breaks isolation and decides to share a code change, he or she is essentially making an assertion that the change has reached a higher level of maturity. This, coupled with the use of local developer builds, helps to ensure that only mature, well-tested code is passed on to the rest of the development team, a primary benefit of continuous integration.

Enable local developer builds

If a primary goal of continuous integration is to improve software quality and reduce downstream build and test failures, then perhaps the single most important SCM best practice is to enable automated local developer builds. To this end, developers should be provided with access to a local build environment that mirrors the production build environment, including

- database schema,
- configuration files,
- environment variables,
- shared libraries and files, and
- known compiler/linker/runtime environments.

This minimizes the “it compiled fine on my machine” syndrome that plagues many development efforts.

Once the build environment is available, every effort should be made to automate the local build, so that each developer can build easily and quickly without having to perform a series of manual steps. Ideally, the developers should have access to a private build area that uses the same tools and configurations as the production builds. As part of configuring a local build, developers should be trained to take several steps to ensure that local changes will have minimal negative effects once shared with other developers and teams. These steps are as follows:

- Updating their private workspace to obtain the latest shared configuration code
- Merging conflicting changes from the shared configuration to their private workspace

By ensuring that developers have known, good build environments, and that they have the latest code with all conflicts resolved, there is high probability that the local build will succeed. If the local build is successful, a continuous integration build that includes the developer’s changes should also succeed.

Frequently update code in the SCM system

Traditionally, developers tend to put off sharing their changes, sometimes for days, because they don’t want to affect other people too early, or don’t want to get blamed for breaking the build. Unfortunately, this strategy tends to backfire and typically leads to more problems in debugging larger sets of changes after a build breaks. Independent of whether continuous integration is being employed, encouraging developers to build, test, and share their code in small “chunks” is a simple and effective way to improve team collaboration and reduce costly broken builds.

The importance of frequently updating the code in the SCM system is amplified when organizations move towards a continuous integration model. Continuous integration represents a paradigm shift for software development, emphasizing communication between developers about what changes have been made. By breaking down tasks into small chunks that take several hours to complete, developers can “commit” or “promote” their changes frequently and receive immediate feedback about the quality of the changes through the continuous integration build and unit test results. As teams get accustomed to this new approach, developers also gain a sense of progress by seeing not only their own changes build successfully, but also seeing the changes that other developers have made available.

Establish a staging and isolation hierarchy
Proponents of continuous integration commonly suggest branching as little as possible and having developers work directly from the mainline as much as possible. However, this approach has several difficulties:

- It puts the stability of the mainline at risk.
- It presupposes that traditional legacy branches are the only available isolation mechanism.
- It decreases the flexibility and agility required for fast iterative development.

With modern SCM systems, a better approach is to implement a staging and isolation hierarchy for the development process. A staging and isolation hierarchy uses objects in the SCM system to represent the dependencies between development groups and process steps. For example, you may wish to model the following teams and activities:

- Release engineering
- Quality assurance
- Product engineering
- Component engineering

Each team or activity is assigned the equivalent of a private workspace (variously called “streams” or “branches” depending on the SCM system). Each team then receives the same benefits of private workspaces that individual developers receive.

With a staging hierarchy, changes move from less stable configurations to more stable as they are tested and deemed “good” for the next level. This allows the code to be stabilized as it gets ready for release without developer downtime. It also allows additional separation for each team if needed, so that the team’s changes can be integrated and tested before the components are integrated together.

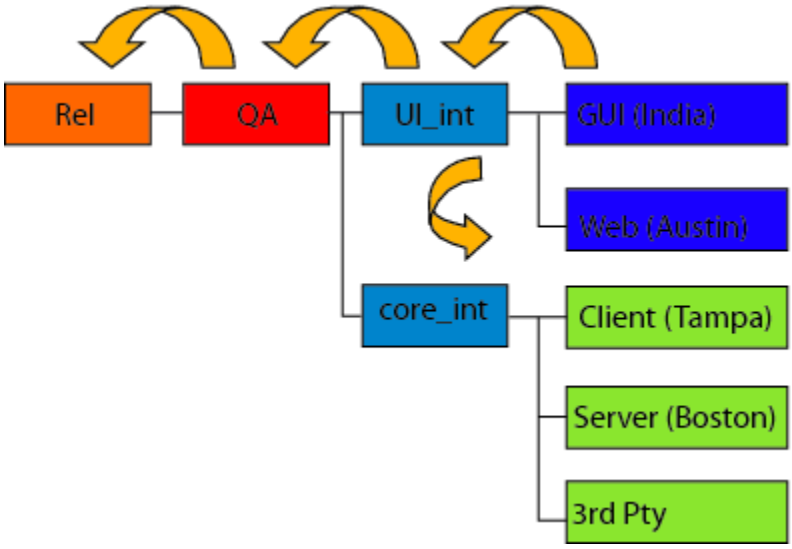

Figure 1: Typical development scenario

In Figure 1, there are four development teams as well as an area for accepting third-party code drops. The teams are located in different geographical areas. The hierarchy represents the normal flow of changes through development from stage to stage. In the example of Figure 1, changes provided by the GUI product engineering team in India flow from individual developer workspaces (not shown for brevity) to the GUI stage, where they can be continuously integrated and tested. Mature changes then flow to the UI_int stage and on to the QA and Release (Rel) stages, again being subject to continuous integration and testing at each stage. The web development team in Austin picks up well-

tested changes from the UI_int stage and uses them as the basis of their development work; when the web changes are mature they can be pushed up the hierarchy and subject to broader testing in the UI_int, QA and Rel stages.

Using a development hierarchy provides more opportunities for checkpointing. Every change introduced into the system is a potential source of failure, and thus a potential checkpoint. If a change proves to be unstable, you can return both the source stage and the destination stage back to a previous checkpoint. By contrast, mainline development only offers you a single opportunity for checkpointing, specifically, the state of the main codeline itself. Unless your development process includes "freezing" the mainline for a long enough period to build, test and otherwise validate, the chances of isolating and checkpointing at an appropriately fine level of code granularity are slim, making any available checkpoints stale and of limited utility.

Automate builds at all stages in the hierarchy

In order to give developers prompt feedback about the changes submitted, the code must be built frequently, ideally several times per day. A continuous integration server such as CruiseControl, CruiseControl.NET or Draco.NET can be employed to automate this process. The continuous integration server periodically polls the SCM system for changes, populates the changes to the build server, initiates the build process, and reports the results of the build and unit tests.

It is important to note here that the continuous integration server utilizes the existing build scripts and build environment to execute the build. For example, if *make* is used to compile and link components written in C, then the continuous integration server will call the makefile to initiate the build process. Because the continuous integration system uses the existing build, it is important for development groups to devote time and effort to

- making the build as fast as possible,
- building automated unit tests and
- including unit tests as part of the build process.

Spending time on these items, even if it involves some rework of the build system to make it more compatible with a continuous integration environment, will improve not only the build process but the overall quality of the software release.

When utilizing continuous integration, it is crucial to communicate the results of the builds to the entire development team. Continuous integration system planners should consider a scalable communications method such as e-mail notification or an internal website to display build results. Continuous integration servers such as CruiseControl come with built-in web reporting that can be easily customized, so that build results can be displayed on LCD panels in common areas at geographically dispersed locations. In this way, team members can easily see and respond to the build results and reduce the "fix latency" often encountered with nightly or weekly integration build approaches.

Summary

Continuous integration, while not a new concept, is rapidly being adopted as a key technology process in software development organizations as part of the shift towards Agile methodologies. Coupled with a robust SCM system and the proven best practices outlined herein, engineering managers, quality assurance managers and developers can use continuous integration to improve software quality, reduce costly rework due to broken builds and ultimately increase the business value delivered to customers.