Utilizing SCM to Scale Continuous Integration
When you are creating new software, what do you think about? Don’t you think about how it will scale to meet your customers’ needs as they grow with high availability? Even if you don’t achieve that on the first try, aren’t you still thinking about it and striving to achieve it? You know that to do it, you will need to make the right technology and architectural choices. Over time, you may need to change some of those choices to keep pace with competitors. Even if your current needs are modest, software development architecture has evolved technologies and patterns to allow software to scale from a single user to hundreds or even thousands of users on multiple platforms at multiple locations with 99.999% uptime.

On the other hand, as an industry, we don’t seem to be very good at scaling up the process of software development itself. Even when what needs to be built is primarily made up of well-understood tasks and requires very little discovery, it seems that when the size of the effort goes beyond 5-10 people, things start to break down pretty rapidly. Adding more people usually doesn’t scale in the same way that adding more hardware does.

I think this stems from the belief that software automates a well-defined problem, but that the process of software development is a creative endeavor and cannot benefit from the same kind of thinking that goes into designing software. However, after spending many years involved in thinking about how to improve the process of software development, experience has convinced me that a great deal of the process of developing software can be treated as infrastructure. This infrastructure supports the creative parts of the software development process, but can be clearly delineated from it.

By thinking about the infrastructure of your process in the same way as you would think about any manual process that requires automation, you can leverage your skills as a developer and apply them in a new way. The process of developing software is, in effect, an algorithm implemented with various technologies. You can think of your team, the technologies you use, and your development methodology as software.

Continuous Integration at Scale

The main goal of software is to automate and simplify what would otherwise be accomplished using a manual process. This gives the users of the software leverage to do more with less. Instead of balancing our checkbook by hand, we can use Quicken to do it faster and more accurately. Instead of maintaining the records of millions of people’s financial transactions with paper and pencil, banks use mainframes.

What technologies are you using? What is the architecture of your organization? Will it scale from its current size to double its size? Will it scale seamlessly to include new teams in new locations? What will happen if you acquire a company?

When creating software, you want to design it so that it is flexible and adapts to new circumstances. The same should be true for your development organization. In this paper, we apply this way of thinking to the problem of scaling Continuous Integration to development efforts of more than 10 people.

Integration

How frequently have you merged your code with changes from the mainline, only to find that the result doesn’t build, or it builds but it doesn’t work? Monthly? Weekly? Daily? Hourly? Or worse, how often have you made changes that broke the build, requiring you to quickly correct the problem while getting flames from your team members?

If everybody in development is making changes against the mainline, introducing a bad change affects everybody until it is detected and corrected. Waiting for someone to fix a change that is impacting everybody is bad enough when the change was committed locally, but it becomes severe when the committer won’t even be awake for another eight hours.

Big-Bang Integration

Integration is tough enough when you are just integrating your work with the work of others in your small team or when a small team is doing everything. But when you are part of a large team, there is also something called “Big-Bang” integration. That’s the integration of the work that multiple teams have been working on for long periods of time. In a typical project, this integration is done in a phase toward the end of the project. During integration, many problems are discovered for the first time, which leads to delays, changes in scope, or both.
**Continuous Integration**

A practice that has emerged to address the problems of integration is called Continuous Integration. The basic idea is that if integrating changes together and getting build and test results on a regular basis is a good idea, integrating and getting build and test results on every change is even better.

With Continuous Integration, all work from all teams is integrated into a single code line as frequently as possible. Every check-in automatically triggers a build and usually a subsequent run of the test suite. This provides instant feedback about problems to all interested parties and helps to keep the code base free of build and test failures. It also reduces the integration headaches just prior to release.

Continuous Integration has been recommended for many years with great success. But it has a dark side as well. The larger and more complex the project, the higher the chance that it devolves into “Continuous Noise.”

For any meaningful project, a full integration build and test will take time. Even if you somehow get that time down to 10 minutes, a lot can happen in 10 minutes, such as other people checking in their changes and invalidating your results (whether you realize it or not). It is more likely that this full cycle will take on the order of an hour or more. While you can certainly do something else during that hour, it is inconvenient to wait and necessitates task switching, which is a well-known productivity killer.

In this case, you get notified every 10 minutes or so (depending on how much building and testing is going on) that the build and test suite is still failing. It may be for a different reason every time, but it doesn’t matter. It is tough to make progress when the mainline is unstable most of the time. Continuous Integration does not cause this problem, but Continuous Integration does expose the problem.

Continuous Integration does not actually reduce the amount of integration required. It doesn't really solve a scaling problem either. There are only two benefits of Continuous Integration. The first is that it forces you to break work down into small, manageable pieces. The second is that it spreads the integration work out over the entire development timeframe instead of just a short period at the end when you have less time to find and fix all of the issues discovered during integration.

The real question is, what is a good way to structure this integration so that it will scale smoothly as you add more people to the equation? A good starting place is to look around for a pattern to follow. What are some similar situations? Everything your organization needs to do in order to produce the best possible development organization can be entirely derived from the patterns and practices at the individual level. This approach makes it much easier to understand, and more will likely follow it.

**Self Integrity**

When you as an individual work on a change, you often change several files at the same time. A change in one file often requires a corresponding change in another file. The reason that a single individual works on those files is because the changes are tightly coupled and don’t lend themselves to a multi-person effort.

![Figure 1. Individual developer pattern](image-url)
As an individual developer, there are two things that you do to shield yourself and others from instability. Although you make frequent changes to your workspace, which means its stability goes up and down rapidly, you only check in your changes when you feel like you've done enough testing of those changes that you've sufficiently reduced the risk of disrupting everybody else that is depending on the mainline. When you check in your changes, you make an assertion that your change has reached a higher level of maturity.

Conversely, you only update your workspace when you are at a point that you feel you are ready to absorb other people's changes. Because other people only check in when they feel the changes are ready and you only update when you feel you are ready, you are mostly shielded from the constant change that is going on all around you.

These two simple practices act as buffers that shield other people from the chaos of your workspace while you prepare to check in and shield you from problems that other people may have introduced since you last updated your workspace.

While it may seem like a bit of a trivial case, you can think of this process as self-integration. This is the basis of Multi-Stage Continuous Integration. If individual isolation is a good idea, then isolation for features, teams, team integration, staging, QA and release is an even better idea.

Moving From Known Good to Known Good
This integration pattern is also found in the interactions of customers and software suppliers and the interactions of software producers and third-party software suppliers. Your customers don't want random builds that you create during development, and you don't want random builds from third parties that you depend on. The reasons are simple and obvious. The level of quality of interim builds is unknown, and the feature set is unknown. Your customers want something with a high level of quality that has been verified to have that level of quality. Likewise, you want the same from your suppliers. Your suppliers include things like third-party software libraries or third-party off-the-shelf software that your application depends on such as databases and web servers.

As with your own software, each third party that you rely on produces hundreds, if not thousands, of versions of their software, but they only release a small subset of them. If you took each of these as they were produced, it would be incredibly disruptive, and you would have a hard time making progress on your own work. Instead, you move from a known good build to a known good build, their externally released versions. Your customers do the same.

This simple principle should be applied throughout your development process. Consider each individual developer as both a consumer and producer of product versions. Also think of them as a third party and of each team as well as each stage in your development process this way. That is, as a developer your teammates are customers of the work that you produce. And you are a customer of the work that they do. You want to move from known good version to known good version of everything you depend on.

It's All for One and One for All
The next level of coupling is at the team level. There are many reasons why a set of changes are tightly coupled. For instance, there may be a large feature that more than one person can work on. As a team works on a feature, individuals need to integrate their changes with the changes made by the other people on their team. For the same reasons that an individual works in temporary isolation, it makes sense for teams to work in temporary isolation. When a team is in the process of integrating the work of its team members, it does not need to be disrupted by the changes from other teams. Conversely, it would be better for the team not to disrupt other teams until they have integrated their own work. Just as is the case with the individual, there should be Continuous Integration at the team level, but then also between the team and the mainline.

Multi-Stage Continuous Integration
So how can we take advantage of the fact that some changes are at an individual level and others are at a team level while still practicing
Continuous Integration? By implementing Multi-Stage Continuous Integration. Multi-Stage Continuous Integration takes advantage of a basic unifying pattern of software development. Software moves in stages from a state of immaturity to a state of maturity, and the work is broken down into logical units performed by interdependent teams that integrate the different parts together over time. What changes from shop to shop is the number of stages, the number and size of teams, and the structure of the team interdependencies.

For Multi-Stage Continuous Integration, each team gets its own branch. The thought of per-team branching and merging may be frustrating, but that’s probably because you are thinking of branches that contain long-lived changes. We’re not going to do that here.

There are two phases that the team goes through, and the idea is to go through each of them as rapidly as is practical. The first phase is the same as before. Each developer works on his or her own task. As he or she makes changes, Continuous Integration is done against that team’s branch. If it succeeds, great; if it does not succeed, then that developer (possibly with help from his or her teammates) fixes the branch. When there is a problem, only that team is affected, not the whole development effort. This is similar to how stopping the line works in a modern lean manufacturing facility. If somebody on the line pulls the “stop the line” cord, it only affects a segment of the line, not the whole line.

On a frequent basis, the team will decide to go to the second phase: integration with the mainline. In this phase, the team does the same thing that an individual would do in the case of mainline development. The team’s branch must have all changes from the mainline merged in (the equivalent of a workspace update), there must be a successful build, and all tests must pass. Keep in mind that integrating with the mainline will be easier than usual because only pre-integrated features will be in it, not features-in-process. Then, the team’s changes are merged into the mainline, which will trigger a build and test cycle on the mainline. If that passes, then the team goes back to the first phase where individual developers work on their own tasks. Otherwise, the team works on getting the mainline working again, just as though they were an individual working on mainline.

The following diagram shows a hierarchy of branches with changes flowing from top to bottom and in some cases back towards the top. Each box graphs the stability of a given branch in the hierarchy over time. At the top are individual users. They are making changes all day long. Sometimes their work areas build; sometimes they don’t. Sometimes the tests pass; sometimes they don’t. Their version of the software is going from stable to unstable on a very frequent basis, changing in the order of every few minutes. Hopefully, users only propagate their changes to the next level in the development hierarchy when the software builds for them, and an appropriate amount of testing has been done. That happens in the order of once per hour or so, but ideally it happens no less than once per day.

Then, just as individuals check in their changes when they are fully integrated, the team leader will integrate with the next level. And when the integration build and test are done, they will merge the team’s changes to the next level. Thus, team members see each other’s changes as needed, but only team members’ changes. They see other team’s changes only when the team is ready for them. This happens in the order of several times per week and perhaps even daily.

Changes propagate as rapidly as possible, stopping only when there is a problem. Ideally, changes make it to the main integration area just as frequently as when doing mainline development. The difference is that fewer problems make it all the way to the main integration area. Multi-Stage Continuous Integration allows for a high degree of integration to occur in parallel while vastly reducing the scope of integration problems. It takes individual best practices that we take for granted and applies them at the team level.

Distributed Integration
All of the reasons that make Continuous Integration a good idea are amplified by distributed development. Integration is a form of communication. Integrating distributed teams is just as important as integrating collocated teams. If you think of your teams as all being part of one giant collocated team and organize in the same manner as described in the section on Multi-Stage Continuous Integration, it will be much easier to coordinate with your remote teams.
Multi-Stage Frequent Integration at Litle and Co.

At Litle, the biggest problem it ran into, as it has grown, was integration of newly developed code into the existing codebase. To address this, in 2007 they added the practice of Multi-Stage Frequent Integration to its implementation of XP. Litle does frequent integration instead of Continuous Integration because a full build or test cycle takes 5 hours.

Prior to implementing Multi-Stage Frequent Integration, Litle would have to manually pore over all build and test failures to determine which change caused which failures. This was done by very senior developers that were familiar with all aspects of the system to be able to understand complex interactions between unrelated components of the system.

Using Multi-Stage Frequent Integration, each project works against its own team branch, merging changes from the mainline into its team branch every day or two, and doing its own build, test, or cycle with just that team’s changes.

Thus, any failure must be a result of one of their changes. When the build or test cycle is successful, the team then merges its changes into the mainline. As a result, any team-specific problems are identified and removed prior to mainline integration and the only problems that arise in the mainline are those that are due to interactions between changes from multiple teams. The isolation also simplifies the job of figuring out what changes may have caused a problem because they only have to look at changes made by a single team, not the changes made by all developers.

Adopting Multi-Stage Continuous Integration

Getting to Multi-Stage Continuous Integration takes time but is well worth the investment. The first step is to implement Continuous Integration somewhere in your project. It really doesn’t matter where. Here’s some recommended reading: “Continuous Integration” by Paul Duvall, Steve Matyas, and Andrew Glover.

The next step is to implement team or feature-based Continuous Integration. Once you have that working, consider automating the process. For instance, you can set things up such that once Continuous Integration passes for a stage, it automatically merges the changes to the next level in the hierarchy. This keeps changes moving quickly and paves the way for easily adding additional development stages.

Multi-Stage Continuous Integration has been successfully implemented in many shops and every time the developers say something like, “I never realized how many problems were the result of doing mainline development until they disappeared.”

Damon Poole was the Founder and CTO of AccuRev. He continues to write frequently on the topic of Agile development and the implementation of Agile techniques that scale smoothly to large distributed teams.

Conclusion

When your organization develops software, your ultimate goal should be for that software to be flexible enough to adapt to new circumstances. To enable this to happen, your development organizations also need to be flexible. This paper has outlined how effectively Continuous Integration can be applied to enable flexible software development.

To find out how Micro Focus® can help your organization to implement AccuRev, visit: www.borland.com/accurev