Crossroads News
A Monthly Publication for Software and CM Professionals

Codeline Merging and Locking: Continuous Updates and Two-Phased Commits

Brad Appleton, Steve Konieczka, Steve Berczuk – November 2003

This month we will discuss the subject of merging our changes with the codeline. First we will outline the basic process to use. Next, we'll mention some of the obstacles that arise, and strategies to overcome them. We will conclude with a discussion of how to decide which strategies are right for you!

Working on a Change-Task

Fantastic "Fred" has been working on an enhancement to add the new "Whiz bang" functionality into his team's software product. He thinks he has it all working now and feels it is ready to be integrated together with the rest of the team's changes on the codeline. To get to this point, Fred already applied the following SCM patterns *[1]*:

1. Fred created a *Private Workspace* in which to implement and test his changes. This isolates Fred's work from the team-wide codeline, and insulates the team from the effects of Fred's in-progress changes before they are working properly.
2. Once the *Private Workspace* was created, Fred populated the workspace with a *Repository View* of the latest stable state of the codeline. This gave him a "snapshot" of the most recent working state of the codebase in which to begin making his changes.
3. Fred checked-out the files he needed to implement the functionality for his assigned development task and went through a few cycles of edit-compile-build-test before he finally seemed to have everything in working order. He performed a *Private System Build* to execute the requisite *Unit Tests*, *Smoke Tests*, and *Regression Tests* to verify that it all seemed to work.

You'll note that Fred and his team are using the classic *copy-modify-merge* paradigm for making their changes to the codeline (see *[2], [3], and [4]*): First, copy the latest state of the codeline into your private workspace; then make your changes; finally, merge the changed result back to the codeline.

Codeline Commit Preparation

At this point Fred has completed the "copy" and "modify" portions of his overall *change-task*. Now Fred thinks he is ready for the last part, the "merge" step. This is the "beginning of the end" of the change-task and is where things start to get interesting. Here is what Fred still needs to do before actually merging the files into the codeline:

Rebase: Ensure that the code in his private workspace is "up to date" with the latest state of the codeline. If it isn't, Fred must perform a *Workspace Update* to populate his workspace with the latest versions of any new files on the codeline (also known as "rebase" or "rebaseline".)

Reconcile: If there were any checkout *collisions* (where the same file was changed both in Fred's change-task and on the codeline) then Fred will need to merge those files into his workspace. If there were any merge *conflicts* (overlapping changes to the same region of code by more than one change-task) then the merge may be non-trivial.

Rebuild & Retest: If any new file versions resulted from the *Workspace Update* (even if they didn't need to be merged), Fred probably needs to perform another *Private System Build* to ensure that he didn't "break" anything and his changes still work with the latest state of the codeline.

Resolve: If the re-build or re-test failed, then Fred must fix the problem and then restart the cycle all over again. If the problem is hard to fix, Fred may wish he'd created a *Private Checkpoint* to capture the state of his previously working changes immediately prior to the *Workspace Update* (that way he can use the checkpoint as a basis for file differencing/comparison, and even "rollback" his workspace if necessary).

Ready-to-commit: Finally, Fred will be ready to perform a *Task-Level Commit* to checkin his changes to the codeline (which hopefully no one else has attempted while Fred was going through all of this commit-preparation work). The *Task-Level Commit* ensures that the set of changes (*change-set*) for a change-task are checked-in to the codeline as a single atomic transaction: either the checkin succeeds for all of the files in the change-set, or it succeeds for none of them.

Two-Phased Codeline-Commit

It is no coincidence that the above appears similar to the “two-phased commit” protocol used by databases to ensure the ACID properties for a database transaction (ACID is an acronym for Atomicity-Consistency-Isolation-Durability). Many version control tools employ “transaction-based CM” mechanisms *[5]*. The standard *Two-Phase Commit* protocol consists of:

Phase 1 – Preparation: For Fred, this is what he was doing in order to ensure his workspace was up-to-date and working properly with the latest state of the codeline. He started with a *Workspace-Update* (which perhaps really should have been preceded by a *Private Checkpoint*) to synchronize with the codeline, then followed-up with a *Private System Build* to make sure it still worked properly.

Phase 2 – **Commit**: This is the actual committing (atomic checkin) of all modified files to the codeline. The *Task-Level Commit* operation commences when the first file in the change-set is about to be checked-in to the codeline; it successfully completes after the last file in the change-set has been checked-in to the codeline.

As we noted above however, it is still possible that another developer may commit their changes during the time that Fred was performing his commit-preparation activities. A *Task-Level Commit* ensures that all files modified for a change-task are committed together all at once as a single operation - it does not prevent a change from being committed while others are committing or preparing to commit their changes.

Concurrent Codeline Commit Contention

This is what happens when more than one developer wishes to merge their changes to the codeline during the same period of time. The "window of opportunity" doesn't just refer to the time when the actual codeline "commit" operation starts and ends. It refers to the period of time in between when a developer first knows their changes are good/stable enough to commit to the codeline (and is about to begin commit-preparation activities), up until the time they finish merging those changes to the codeline.

Suppose our “Fantastic Fred” has finished making all his changes and has successfully committed them to the codeline. We neglected to mention Fred’s other teammates: Barney and Betty. Both Barney and Betty have also been busy with their own change-tasks. When Fred finished committing his changes, they both tried to be next to commit their changes. Both tried to execute their commit-preparation activities as quickly as possible. Betty beat Barney and committed her changes first!

So now poor Barney needs to keep re-checking and re-updating his workspace because the codeline he is trying to commit to is a moving target. Barney hates trying to hit a moving target and wants to devise some tactics to prevent this situation. He has two ideas about how to do this:

a) Perhaps some rule or “locking” mechanism can help ensure that both phases of the codeline-commit process are treated as part of a single atomic transaction.
b) Or maybe Barney just needs to find a way to decrease the amount of time consumed by his commit-preparation activities.

Barney feels inclined to investigate both of these alternatives. He decides to first see what he can do about going faster before looking into ways of slowing others down (e.g., locking).

Conquering Codeline Commit Contention

So what can Barney do to go faster and minimize the time spent doing commit-preparation? What are the major contenders for the main bottlenecks in the commit-preparation process? They seem to be the following:

- Time spent merging and reconciling any conflicts as a result of doing a *Workspace Update*
- Time spent doing the *Private System Build* to rebuild/retest changes after the *Workspace Update*

Barney talks with his local build-gurus and testing gurus to see what can be done about improving build-time and testing time. They do what they can to help him automate as much as possible of the build & test activities. He also consults some IT hardware/network experts and build-file gurus to see if any hardware or software can be thrown at the problem to make builds go faster, and has the build scripts optimized.

Barney also considers using an incremental build (instead of a full build) in most *Private System Builds*. This could reduce rebuild times considerably, at the expense of decreasing the thoroughness of build-feedback. To compensate, full builds could be automated to occur at least daily (or more frequently).

Barney's efforts thus far have helped not only Barney - they helped the entire team too! (Thanks Barney!) Unfortunately for Barney, this also helped make Betty go faster too and she still beats Barney to the punch.

So Barney decides that if he can reduce the likelihood that his workspace's *Repository View* becomes "stale" (out-of-date with the codeline), that will "nip the problem in the bud": if his workspace is already "fresh" with the latest changes from the codeline, then the *Workspace Update* he performs at the beginning of commit-preparation will simply be a "sanity check" and will have no effect because most of the time he is already up-to-date.

If there are no new codeline changes to update his workspace, then not only does he avoid having to spend his time merging, he also avoids having to spend time rebuilding and retesting the updated workspace.

Continuous Workspace Update

Barney decides to do everything in his power to keep his workspace up-to-date as often as possible. He will update his workspace "continuously": immediately after anyone commits a change to the codeline, Barney will update his workspace. At least that's the theory. In reality, Barney sometimes needs a few extra minutes to finish what he was in the middle of before updating his workspace.

As he starts adopting this habit, Barney finds it would be exceedingly convenient to know whenever a new change is committed to the codeline, and to be able to checkpoint his changes before updating his workspace (for the sake of posterity, and his own posterior):

Post-Commit Notification: Barney needs some kind of indicator or notification whenever a new change is committed to the codeline. Otherwise he would need to continually poll for this information. He also would like to see which files were modified by the newly committed change (either as part of the commit notice, or as part of some codeline "commit-query" mechanism).

Many version-control tools provide event-based notifications out-of-the-box. Unfortunately for Barney, his tool isn't one of them. But this is easy to implement with a simple "commit-wrapper" script, or using a post-commit "hook" (or '"trigger") that executes the script he specifies. So Barney implements a simple subscription mechanism that tracks a list of email addresses to notify for commits to the codeline. (Barney subscribes himself using his "Instant Messaging" email address.)

Right after Barney implements this, Betty mentions something to him: "Since we're all sitting within earshot of each other, why not simply make a verbal announcement when we commit a change!"

This is even simpler than the solution Barney just implemented! Barney decides to stick with his implementation anyway. He thinks to himself: "It's still convenient to see the list of files committed; and occasionally some team-members work from home or from a remote office location."

Private Checkpoint: Before he updates his workspace, Barney would like to make a snapshot of the current state of his workspace to use as a basis for comparison and "rollback" of the to-be-updated file versions. He considers three possible methods for implementing *Private Versions* in order to create a *Private Checkpoint*:

Private Archive: He could simply create a "`checkpoints`" subdirectory in his private workspace. Whenever he needs to checkpoint his changes, he can copy the files he wants to checkpoint. He could use the "query" flag of his tool's `update` command to obtain the list of to-be-updated files and automatically copy them, along with the files he has changed, to a named/numbered subdirectory in his private archive.

As simple and clever as the private archive seems, Barney can't help feeling that it "reinvents the wheel" to do versioning capabilities outside of the version-control tool when the tool itself already knows how to do this (and exists for this very purpose). Plus, the codebase will have no record of the private versions he created: if any "accidents" happen in his private workspace, or if he wants to compare against checkpointed private versions for historical purposes (after his task is complete), his checkpointed changes will be lost or unavailable for use.

Private Branch: He could create a branch upon which to do his development work: He can checkout files to his *Private Branch*, and when he wants to checkpoint his changes, he can checkin to his branch. The resulting versions will be written to his *branch* instead of the codeline, thus making the versions "private." When he is ready, the *Task-Level Commit* will checkin the *Private Versions* to the codeline.

This approach has the advantage of using the version-control tool to do the private versioning. Though the versions on the *Private Branch* aren't in the codeline, they <u>are</u> in the codebase. Using this branch does <u>not</u> increase the amount of time between when his files are checked-out from the codeline and then committed back into the codeline; what it does is provide a "safe haven" for him to create intermediate versions that are useful to him, but which would otherwise break the codeline if checked-in before his change was "safe" to commit.

Task Branch: This approach is essentially a special case of a *Private Branch*. Whereas a *Private Branch* may continue to be used for successive development change-tasks, a *Task Branch* is used only for the duration of a single development task (and is typically "locked" against further changes once the task is completed). In addition to providing *Private Versions*, the *Task Branch* also serves as a *Change Package* that bundles together all the versions and modifications for a particular change-task.

Barney thinks this is "nifty" but he doesn't require the additional "change packaging" capability. (Several version-control tools already provide their own mechanism for this.) In his case, he doesn't feel the additional overhead of creating a new branch for every task "buys him" anything new that he doesn't already get with a single *Private Branch* instead of multiple *Task Branches*.

Some of you reading this may have noticed an important difference between *Private Archive* and *Private/Task Branch*. The set of checkpointed files is different between the two approaches:

- As described above, using a *Private Archive* checkpoints all files in the *Private Workspace* that are to be updated by *Workspace Update*.
- Using a *Private/Task Branch* checkpoints only those files that Barney modified in his workspace on his branch. Any files that he did not modify, but which will be updated from the codeline, are not checkpointed.

This difference is often not very significant, but sometimes it can be. If it is going to be an issue in your environment, the `checkpoint` mechanism requires a bit more implementation effort:

Checkpoint Label: The two different file-sets can be computed (perhaps as an automatic part of the *Workspace Update*) and their union can be tagged as the complete checkpoint-set. This may use the version-control tool's built-in label/tag operation, or it may simply capture the file names and their versions in another file, or in a named attribute/property associated with the workspace or branch (or even in the comment-text for the checkin).

In the end, Barney decides to supplement his practice of *Continuous Workspace Update* with *Post-Commit Notification* (implemented via a hook that emails a local mailing-list) and with *Private Checkpoints* (implemented via a *Private Branch* to enable the use of *Private Versions*, but without creating a *Private Label*).

Using the *Continuous Update* pattern helped Barney drastically reduce the amount of time it takes for commit-preparation activities. It also reduced the amount of time it takes Barney to do the *Workspace Update* itself (since he does them incrementally, instead of all at once at the end in "big bang" fashion).

Continuous Update Complements Continuous Integration!

Now that Barney has instituted the use of *Continuous Updates* within his team, it seems his teammates are committing their changes faster and more frequently than before! This appears to be due to two things that he expected, and one thing that he didn't expect:

- Less time spent merging and reconciling any conflicts as a result of doing a *Workspace Update*
- Less time spent rebuilding/retesting changes after the *Workspace Update*
- Less time spent during the interval between when his files are checked-out from the codeline and then committed back into the codeline

That last one surprises him! Usually people are concerned that using *Private Branches* or *Private Versions* will make folks less likely to commit their changes to the codeline as frequently as they should. But Barney observed the opposite effect!

Barney thinks this is because the real problem was that folks were "afraid to commit" before. They were sometimes wary of committing their changes earlier for fear of breaking the consistency of the codeline. Because of this, they would do more work than they really should have in a single change-task before committing their changes.

Instead of encouraging fear-laden "bad practice", the use of *Private Versions/Branches* with *Continuous Update* encouraged them to "sync up" with the codeline more frequently and to checkin files when they previously wouldn't have checked-in anything at all:

> Rather than preferring checkpoints to committing changes, they instead created checkpoints where they would have otherwise done nothing. More frequent merging of smaller changes within a safe, privately versioned environment made them more comfortable with merging. They were no longer "afraid to commit" and began committing their changes with greater frequency and confidence.

The result was that change-tasks were "right sized" to be more compact and more cohesive sets of modifications. The smaller and simpler change-tasks were integrated more frequently back into the codeline! *Workspace Updates* became smaller and easier too (and also more frequent).

Codeline Locking Strategies

After seeing what great success Barney had improving the commit-preparation "cycle time", Betty decided to try her hand at improving the safety and reliability of the *Two-Phased Codeline-Commit* protocol. This implied implementing or enforcing some kind of formal or informal codeline-locking strategy. The first thing she did was investigate how to make sure that the whole team follows the protocol:

Pre-Commit Validation: Before anyone attempts to commit, a query is made to see if Betty's workspace is already up-to-date with the latest state of the codeline. If it isn't, then the commit either fails, or else her workspace is automatically updated. The validation might also automatically rebuild/retest to check for any failures before allowing the commit to proceed (in which case the validation step essentially performs all the commit-preparation activities). Many version-control tools provide event-based validation "hooks" out-of-the-box to script this (otherwise a commit-wrapper script may be used).

A *Pre-Commit Validation* can also initiate a locking protocol of some kind either before or after the commit-preparation activities.

Betty decides to implement *Pre-Commit Validation* so that it checks for and performs a *Workspace Update* (if needed) and initiates a locking protocol. She conducts her own research and found that the following *Codeline Locking* mechanisms commonly recurred in practice (listed in order of increasing formality and complexity):

Single Release Point: There is a single dedicated codeline integration work-area with a single monitor and keyboard (e.g., an integration machine/workspace). This lone integration resource is time-shared (used for only one person and change at a time), thereby ensuring commits are serialized.

Integration Token: Developers might be allowed to integrate their changes from any machine or workspace, but must first obtain a "token" to do so (and only one such token is available). The token might simply be a purely symbolic or physical artifact (e.g., a coin, a key, a rubber chicken or a purple barney-doll), or even something as informal as a verbal "catch phrase" like: *"I've got the integration token!"* being yelled out to the rest of the team. Or it might be as formal as implementing an exclusive lock on a specific file or object outside the codebase.

Codeline Write-Lock: Formally "lock" all or part of the codeline by having the version control tool prevent checkin to the codeline by anyone but the lock-holder (the person who "acquired" the codeline-lock for exclusive write-access). The lock-holder automatically relinquishes the lock after they are done committing their changes. They may also voluntarily release the lock anytime before then if they feel they need more time to reconcile any conflicts or failures since their last update.

Any implementation of a *Codeline Write-Lock* should include a mechanism for querying the lock to see who holds it, and a "backdoor" mechanism for a codeline "owner" or "administrator" to release the lock in urgent circumstances. Sometimes an automatic or default "timeout" is used to ensure the lock is never held for more than a predetermined maximum period of time.

A *Codeline Write-Lock* is typically implemented in one of two ways:

Full Codeline-Lock prevents the entire codeline (all files) from being written (checked-in) by anyone but the lock-holder. It is often implemented via a pre-checkout "trigger" or "hook" that looks for a "locked" attribute/property that is somehow associated with the codeline.

Partial Codeline-Lock prevents codeline-checkin for only those files that are checked-out (were modified) by the lock-holder. This allows concurrent committing of non-overlapping changes at the expense of increasing the risk of an undetected conflict between change-tasks that had no files in common.

The implementation typically uses a reserved (exclusive) checkout from the codeline of each file in the change-task. If any of the files cannot be checked-out than the lock attempt fails and any checked-out files must be unchecked-out.

Those are the two most commonly used approaches. Betty also noticed some less frequently used (but not uncommon) approaches that were basically some mixture of these two:

Double-Checked Codeline Locking employs a full codeline-lock solely for the duration of time necessary to obtain a partial lock. The full lock is released once the partial lock is acquired.

Phased Codeline Locking employs both full and partial codeline locking across the different phases of the two-phased commit cycle.

- First it obtains a full lock (just before the *Workspace Update*), and if no updates were needed, or if there were no file collisions (and hence no merges), then the full lock is held until the codeline-commit is completed.
- If there were any collisions (or, alternatively, any conflicts) then the full codeline-lock is automatically released (or else automatically downgraded to a partial lock).

At any time during a *Phased Codeline-Lock*, the lock-holder may explicitly "promote" or "demote" the lock (from full to partial, or partial to full), or explicitly relinquish the lock altogether. This lets the lock-holder decide what they feel is the appropriate balance of "safety" and "productivity" for the nature of the changes and conflicts encountered.

Some version-control tools automatically provide *Codeline Write-Lock* capabilities, and/or allow it to be automatically configured. (For example, ClearCase UCM employs partial codeline locking for it's `deliver` operation). Others provide pre-commit "hooks" or "trigger" to let their users "tool them up" on their own for their own projects.

Betty carefully evaluates the benefits and consequences of each of these approaches, both formal and informal:

- She feels the *Double-Checked* and *Phased Codeline Locking* approaches are "overkill" for the needs of her small project and (mostly) co-located team. They might make more sense for a much bigger project with more people or more geographically dispersed team members where codeline contention is much higher and the codeline is very volatile.
- Betty appreciates the simplicity of the *Single Release Point*, but feels it is a tad too crude for her taste. She thinks it just a little "primeval" to restrict all integration to take place in a single physical location away from where everyone normally sits. (Plus it won't work very well for the occasionally remote team-member.)
- Betty likes the simplicity and informality of the *Integration Token* approach and feels it might work well for her particular small team and project. If they turn out to not be disciplined enough in its use, she figures she can easily "escalate" the formality to use a *Full Codeline-Lock*. And if allowing only one commit at a time ends up being too much of a bottleneck, then the *Partial Codeline-Lock* still seems relatively simple and straightforward enough for her needs.

After considering all of these factors, Betty opts for using an *Integration Token*, with *Full Codeline Locking* and *Partial Codeline Locking* as her contingency-plans if the token-based approach doesn't pan out.

Conclusion: Selecting Patterns and Strategies to Use (Its All About Context!)

Some of you may have disagreed with some of the choices made by Betty and Barney. They did what they thought was most appropriate for their project's corresponding complexity and risk-tolerance and their team's size, people, and organizational culture. You will need to do the same for yourself! And your choices may differ based upon important differences in the context of your project's team, architecture, and organization.

So how do we do that? How do we decide which patterns and strategies to use that work for us (rather than Fred, Barney, and Betty). We need to look at our context and our needs and match them to the applicable context of the patterns and strategies. And we need to look at the tradeoffs between the competing forces or concerns that result from applying a particular solution. Some guidelines follow:

Workspace Update should be used at a minimum immediately prior to committing your changes to the codeline. In this context it is actually part of the commit-preparation activities that also include *Private System Build*.

Continuous Workspace Update should be used to minimize commit-preparation "cycle time" when you want your workspace to always be as "up-to-date" as feasible (even when you are in the middle of a change). If your codeline is not particularly volatile or if build and test cycle times are already fast enough so that you rarely (if ever) have to try and hit a "moving target" when integrating your changes to the codeline, then it may make sense to update your workspace less frequently.

Post-Commit Notification should be used as a tooling mechanism to facilitate *Continuous Workspace Update* or if needed to improve communication about committed changes within the team. Note that *Integration Token* communicates both the start and finish of a commit.

Private Checkpoint/Versions should be used when you want to be able to capture the known state of a change-task in your private workspace. The known state should be one that is not suitable for committing to the codeline based on its *Codeline Policy* (otherwise you should commit your changes instead of checkpointing them). Usually this will be immediately prior to a *Workspace Update* when you want to be able to rollback your workspace if the update "goes sour" and/or use the checkpoint as a basis for file comparison/differencing/merging.

Our experience is that unless your version-control tool does this automatically (or can be configured to do so without adding any development "friction") then people usually don't feel it is necessary - right up until the first time they get burned by it (after which they may tend to do it religiously ☺). It can be handy from time-to-time however.

There are several possible strategies for implementing *Private Checkpoint/Versions*, each with their own set of trade-offs:

Private Archive is appropriate when your change-task is already in progress and you didn't have sufficient reason to believe you would likely need to checkpoint your changes when you started this particular task. It is also suitable when the set of files modified is very small and/or all in one central location within the source-tree (typically all in one subdirectory).

Private Branch is appropriate when you are just starting your change-task and you believe there is sufficient reason to believe that you will likely want to checkpoint your changes at least once before committing them to the codeline. (And you expect to be modifying more than just a handful of co-located files in the source-tree.)

Task Branch is appropriate when working on a change-task that is experimental, or which must not have any "partial functionality" committed to the codeline (only the full and complete functionality). It is also useful if your version-control tool has no built-in mechanism for identifying *Change-Sets* or *Change-Packages* and you want to be able to group together and easily identify the set of changes made for your change-task.

Checkpoint Label is appropriate when you want to checkpoint the state of your workspace (not just the files you changed, but all the files that are updated) and you aren't using *Private Archive*; or when you want to be able to "remember" more than just the most recent checkpoint for the purpose of rollback or as a basis for file differencing/merging.

Two-Phased Codeline-Commit is really an *emergent pattern*. Its use automatically emerges as a result of appropriate use of *Workspace Update* and *Private System Build*. These two patterns in succession (along with pre-update *Private Checkpoints*, other pre-commit activities) simply become a natural part of *Codeline-Commit Preparation* that defines the window of opportunity in which codeline commit-contention may occur.

Pre-Commit Validation should be used whenever you need to impose a more formal (and automatic) team-discipline of following the two-phased commit protocol and ensuring that "stale" workspaces never have their changes committed to the codeline. It can also be used to ensure that tests have been run to ensure the committed change won't "break" the codeline.

Codeline Locking of some form another (even if it's not a true/formal lock) should be used whenever codeline-commit contention is a legitimate issue, or if there is good reason to believe that it will be an issue. To determine this will require examining the following factors:

- Team size: The larger and/or more geographically dispersed the team, the more rigor and formality is required to keep integration coordinated
- Build/Test time: The longer it takes to build and test, the greater the likelihood of commit contention
- Parallel tasks: The greater the number of parallel tasks, the greater the likelihood of commit contention
- Likelihood and cost of non-collision conflict (when two independent changes don't touch the same files but cause a build failure when checked-in together)
- Contention time-window: The typical expected time-span (duration) for performing *Codeline-Commit Preparation* and *Task-Level Commit* activities
- Completion overlap: The likelihood of multiple tasks being completed within the contention-time-window (especially near the end of an iteration or a release)

Another driving factor becomes minimizing the duration of the contention time-window. This can be accomplished by automating tests, and by utilizing incremental builds to a large extent. Then frequent (continuous) workspace updates reduce the likelihood of the workspace being out-of-sync with the codeline.

If you decide locking is called for, then the locking strategy should ideally be part of the *Codeline Policy*. When determining the right locking strategy for your project, the heart of the issue is the level of formality of the locking. Ask yourself: Is it merely by social convention? Is it a little bit stronger than that (like a "token")? Does it need to be more formally enforced and communicated?

If you decide to use an informal locking strategy, then choose between *Single Release Point* and *Integration Token* as follows:

Single Release Point is good for small co-located teams when codeline contention common but not very frequent. A single release area and/or single integration machine effectively locks the entire codeline against checkin by serializing the use of the sole resource with which codeline checkins are made. The scope of the lock is the entire codeline, so if change-set overlap is rare but completion-overlap is common, this may not be the best approach.

Integration Token works well if more explicit team-wide communication of commits is desired, or if the team is not co-located. An informal "verbal" or "visual" token can work well for co-located teams. The *Integration Token* can be used with or without a *Single Release Point*. When it is infeasible or inconvenient to restrict integration activities to a single physical location or resource, then use an *Integration Token* without mandating a *Single Release Point*.

Codeline Write-Lock is appropriate for formally enforced codeline locking. If you decide a formal lock is called for, then the issue of granularity comes into play: Do I lock the whole thing for the whole time? Do I lock just my files for the whole time? Do I lock just my files for part of the time? Or do I perform some combination thereof? There are several formal locking solutions to choose from, each with their own benefits and drawbacks:

Full Codeline-Lock is usually the easiest to implement, but is also the most pessimistic since it locks the entire codeline. Use it when formal locking is desired but completion-overlap is rare and non-collision conflicts are a serious concern (due either to risk or cost).

Partial Codeline-Lock is the most optimistic of the formal locking strategies but is not as simple or easy to implement as full locking (unless the version-control tool already supports it). Partial locking is ideal when completion-overlap is common (particularly near the end of an iteration/release).

Double-Checked Codeline-Lock is appropriate when commit-contention and completion overlap are both common and when the set of modified files is non-trivial in size or takes a non-trivial amount of time to checkout (long enough so that contention even just during the time it takes to obtain the lock is a legitimate concern). Unless your version-control tool implements this automatically, it is typically the most effort to implement of the formal locking strategies discussed here.

Phased Codeline-Lock is appropriate when completion-overlap is common during *Codeline Commit Preparation* activities but not during Task-Level Commit, and non-collision conflicts are a serious concern (due either to risk or cost).

References

[1] **Software Configuration Management Patterns**, by Stephen P. Berczuk and Brad Appleton; Addison-Wesley, November 2002 (see http://www.scmpatterns.com)

[2] *Configuration Management Models in Commercial Environments*, by Peter Feiler; **SEI Technical Report CMU/SEI-91-TR-7**, March 1991 http://www.sei.cmu.edu/legacy/scm/abstracts/abscm_models_TR07_91.html

[3] **Version Management with CVS**, by Per Cederqvist et.al.; http://www.cvshome.org/docs/manual

[4] **Version Control with Subversion**, by Ben Collins-Sussman et.al.; Revision 7480, 2003-10-22 at http://svn-book.redbean.com

[5] *Transaction-Oriented Configuration Management*, by Peter Feiler and Grace Downey; **SEI Technical Report CMU/SEI-90-TR-23**, November 1990 http://www.sei.cmu.edu/legacy/scm/abstracts/abstransaction_cm_TR23_90.html

For those desiring a handy overview of the SCM Patterns from *[1]*, a single page “quick reference” card is now available for download from http://www.scmpatterns.com/refcard.

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. 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*