The Importance of Branching Models in SCM

Among the branching models used in software configuration management, the branch-by-purpose model offers better support for parallel development efforts and improved control of both planned and emergency software releases.

Chuck Walrad
Davenport Consulting

Darrel Strom
Expert Support

If you want to improve software quality, you must first understand your software. What are its pieces? How are they organized and related to one another? If you do not understand your code base, your odds of updating it without breaking something are poor.

All too often, we see projects brought to their knees while trying to get the software out the door to testing groups or to customers for acceptance testing. The developers have worked feverishly to get the features in, and the testers are waiting to test, but the process fails at the integration point. The software components don't hang together, and some components may be missing. Wrong versions get distributed, and previously fixed bugs somehow reappear.

Why don't organizations have a better handle on their software? Is the problem a function of code size and complexity, is it inherent in parallel development efforts, or is it simply a result of staff turnover or cutting corners to meet schedule pressures?

All of these factors can contribute to the situation, but the real problem lies in a fundamental misunderstanding of software configuration management as it applies to real-world application development.

SCM DEFINED

Software configuration management serves two different functions:

- Management support for controlling changes to software products. This function includes the activities classically associated with SCM[1-3]—specifically, identifying the software components, controlling changes to them, recording and reporting component and configuration status, and conducting audits and reviews.
- Development support for coordinating file changes among product developers.[4,5] These activities include file version identification, software building, and release management.

Branching is integral to version management, software build correctness, and release management. It enables parallel development of a new system and provides concurrent support of multiple releases by labeling each instance of a branched configuration item and establishing a mapping between the label and the module revisions, as described in the "SCM Glossary" sidebar.

Good decisions about when and why to branch can make it much easier for developers and release engineers to coordinate software product changes. The right branching strategy makes it easier to deliver the right code, re-create past releases, and—if necessary—roll back to a previous release.

Adopting the right SCM branching model facilitates rapid development, increases overall product quality and process efficiency, reduces the incidence of software failures, and improves organizational performance.

THE BRANCHING MODEL

A branching model embodies the rationale

0018-9162/02/$17.00 © 2002 IEEE

SCM Glossary

Some basic definitions are helpful in developing a fundamental understanding of software configuration management in the real world of application development.

Baseline. In software development, the IEEE standards define a baseline as a "specification or product that has been formally reviewed and agreed upon, that serves thereafter as the basis for further development, and that can be changed only through formal change control procedures."[1] Alternatively, a baseline can be described as a "set of software items formally designated and fixed at a specific time during the software life cycle," or it can "refer to a particular version of a software item that has been agreed upon. In either case, the baseline can only be changed through formal change control procedures."[2]

Branch. "A branch is an agreed upon split of an item [item, product, or system] into multiple iterations [identifying each] ...instance of item, product, or system, ...[providing] an exact mapping between a version label and module revisions."[3]

Configuration management or software configuration management. (1) "SCM involves identifying the configuration of the software at given points in time, systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the software life cycle. The work products placed under SCM include software products that are delivered to the customer and the items that are identified with or required to create these software products."[4] This includes the tool chain used to create, test, and maintain the product. (2) "A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements."[1]

Release. The distribution of a software configuration item outside the development activity. This includes internal releases as well as distribution to customers.[5]

Release engineering. The process of moving a configuration through the software life cycle and delivering the finished product in the appropriate format and media.

Release management. Identification, packaging, and delivery of a product's elements, such as the executable, documentation, release notes, and configuration data.[5]

Software configuration item. An SCI aggregates software designated for configuration management and treats it as a single entity in the SCM process.[1]

Version. (1) "An initial release or re-release of a software configuration item, associated with a complete compilation or recompilation of the SW configuration item."[1] (2) "An initial release or complete re-release of a document, as opposed to a revision resulting from issuing change pages to a previous release."[1] (3) "A particular identified and specified software item."[5]

References

1. *IEEE Std. 610.12-1990, Standard Glossary of Software Engineering Terminology*, IEEE Press, Piscataway, N.J., 1990.
2. *Software Engineering Body of Knowledge*, trial version, IEEE Press, Piscataway, N.J., 2001, p. 108.
3. M. Ben-Menachem, *Software Configuration Guidebook*, McGraw Hill, Maidenhead, Berkshire, UK, 1994.
4. M.Paulk, et al., *Key Practices of the Capability Maturity Model, Version 1.1,* SEI-93-TR-25, Software Eng. Inst., Carnegie Mellon University, Pittsburgh, 1993.
5. *Software Engineering Body of Knowledge*, trial version, IEEE Press, Piscataway, N.J., 2001, p. 111.

adopted for replicating a configuration item—whether a program module or subsystem—into multiple instantiations, each of which bears its own unique and appropriate configuration-identification label.

Selecting the appropriate branching model lets the release engineer serve several masters that sometimes have conflicting interests or priorities: the development group, the testing group, and the support group—which represents the product's end users. To determine the adequacy of a branching model, we evaluate its ability to:

- maintain a stable base for new development by supporting nightly or continuous integration via builds from the top,
- deliver emergency releases—which contain all necessary fixes and no other changes—to testing and to customers,
- test releases that contain all the necessary fixes and no other changes,
- minimize the impact of emergency releases on new development efforts,
- roll back to a previous production release if necessary,
- support multiple sequential versions in the field, and
- support multiple concurrent versions—such as alternative versions for different platforms or different customers—in the field.

BRANCH-BY-RELEASE MODEL

Conventional wisdom and existing standards tell us to manage a software configuration as a series of successive baselines.[2,3] The *branch-by-release model* of code management instantiates this approach. In this conventional model, the code branches upon a decision to release a new version of the product. The new branch serves as the baseline for continuing development. As Figure 1 shows, the old branch contains the released version—the actual historical baseline reference point. That branch is left behind to wither.

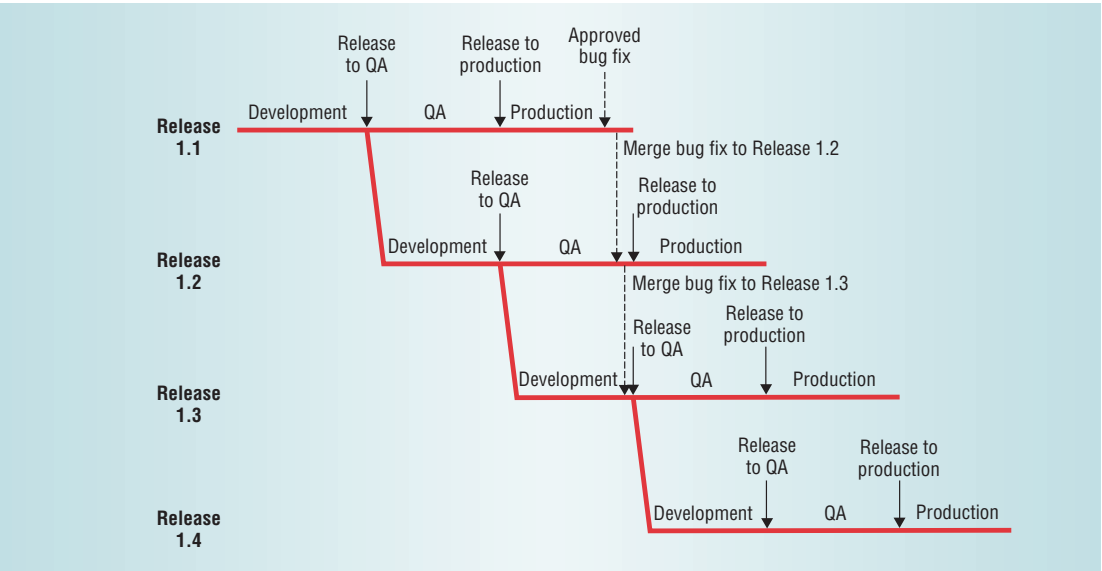

Figure 1. Branch-by-release model. In this conventional model, the code branches when the release engineer delivers a new release. The new branch serves as the baseline for continuing development while the old branch—the code's baseline historical reference—is left behind.

The branch-by-release model appears to provide the series of successive baselines that SCM conventionally requires. It provides a common base for developers to use in making further changes to the code. However, it has two important drawbacks:

- it generally requires serial changes to the code such as sequential check-ins and check-outs, rather than parallel development; and
- it adds complexity and overhead to the support of released versions.

The branch-by-release model is easy to understand—before postrelease bug-fix releases come into play, at least. But making a fix to an earlier release poses many opportunities to lose approved bug fixes or other changes. As long as any version of the product remains supported in the field, the need for a bug fix—and thus an emergency release—remains possible. In such cases, developers must make the fix in that *old* branch and create a new release from it. That's pretty straightforward, but complications arise. Developers must propagate the bug fix down through each subsequent branch to ensure that the bug doesn't reappear in later releases, where it has never been fixed in that release's codeline.

Isolating specific changes and confirming the need to propagate each one to all downstream releases creates an added communication and coordination burden. The environment changes constantly as developers move from one development line to another with each new release. Nor does the model support longer-term development parallel to a release cycle—all code checked out must be checked back in prior to release.

If developers do not check all code back in prior to release, the model does not allow building from the top of the code after release. When the developers check in additional changes after the release, the release engineer must make sure that those untested changes do not find their way into any subsequent new versions, such as emergency releases, created from that line of code. This is particularly problematic because not building from the top drastically increases the risk of incorrect builds for emergency releases. It also undermines the whole baseline concept, because changes never released to the outside have now been introduced in the baseline code.

Finally, the branch-by-release model does not provide a straightforward way to release and maintain multiple concurrent versions in the field.

BUILD-BY-BUG-NUMBER SYNDROME

Using the branch-by-release model leads to the dreaded *build-by-bug-number syndrome*. This occurs when code has been checked in to the old branch after the release, so that the code on the branch no longer matches what was released. The release engineer must handpick the bits of code associated with specific bug fixes that project management or the organization's change control board has decreed necessary for a release. This situation usually arises at the worst time—when the company needs an emergency release. This pressures the team to quickly produce a fix that will remedy an urgent situation.

Tedious at best, and often arduous, the build-by-bug-number process challenges the release engineer to ensure that the release build includes only the pinpointed fixes, all the bits of code needed for each fix, and no other changes. This process often requires the sometimes unwilling participation of the development team, increasing the number of people involved in a tense situation.

How did the project get to this point? Often, the project's management team has determined that changes originally intended for a release could not be included without undue risk. For example, the change might have been seen as potentially destabilizing and there wasn't sufficient time to test it.

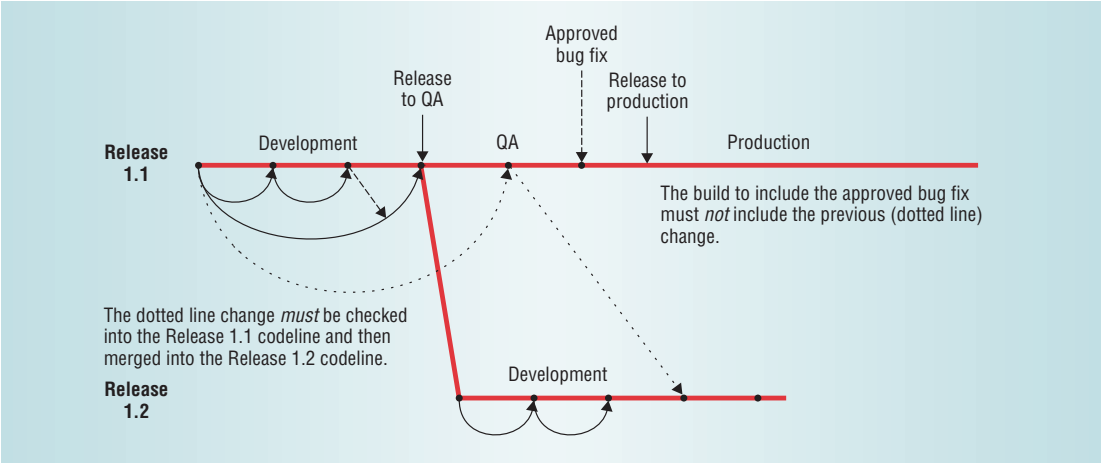

Figure 2. Build-by-bug-number syndrome. This phenomenon occurs when the release engineer must handpick bits of code associated with specific bug fixes, then ensure that only those bits needed for the fixes make it into the release build.

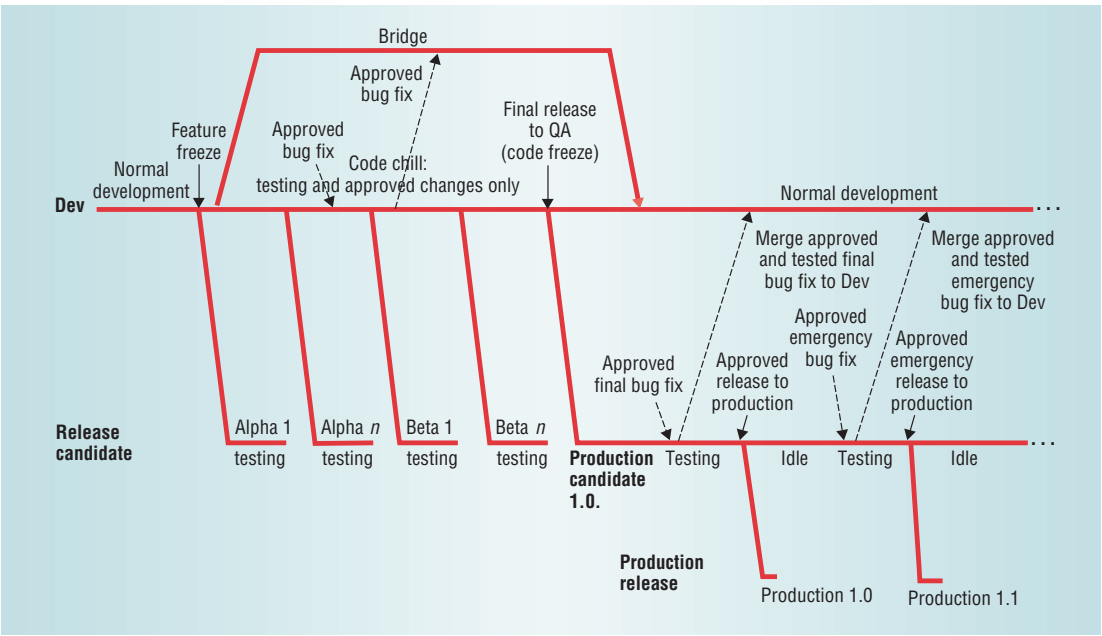

Figure 3. Branch-by-purpose model. The release engineer spins off new branches for specific purposes such as alpha and beta testing, but development work remains on the main development line.

Alternatively, the developers simply could not complete the desired changes in time.

Figure 2 shows the dilemma developers face when confronting this situation. Version control tools require that checked-out code must be checked back in to the same code line from which it was originally taken. If developers do not check the code back in prior to the software product's release, they cannot apply that code directly to the code line being prepared for the next release. The lagging check-in must first be checked into the line it came from, then migrated to the later code line.

Another option requires the developer to abandon those changes and start again on the new development line. Obviously, the developer will oppose this course and the lost work it entails.

As a result, any emergency releases made for that old line must resort to building-by-bug-number. To ensure that only the necessary bug fixes and no other changes—such as lagging check-ins—appear in the emergency release, the release engineer must handpick the specific bits of code associated with the fix. This approach precludes building from the top because doing so would pick up stray changes. Obviously, this scenario invites failure.

In some cases, when poor-quality code requires many postrelease bug fixes, it can lead to abandonment of a disciplined release process in favor of *slipstream* development, as described in the "Slipstream Development" sidebar.

IMPROVED CONFIGURATION MANAGEMENT MODEL

In the *branch-by-purpose model*, shown in Figure 3, release engineering bases the decision to branch on the need to satisfy a specific purpose. Generally, that purpose involves releasing the software and its associated elements, such as documentation, outside the development group. These releases typically mark significant project milestones, such as release to QA for alpha (system) testing, release for beta testing, and so on.

The branch-by-purpose model supports regular releases by design, along with controlled emergency

Slipstream Development

In-house application development groups in information technology organizations often fall into *slipstream* development mode, rather than following a discipline of full product releases. When this happens, the notion of baseline becomes inverted. For one thing, there are no declared product releases. Rather, the *release folders* that these IT groups use may contain just snapshots of the code at particular times, rather than true releases. These snapshots attempt to capture the application's state to establish source code baselines for performing further changes and syncing up the development code line with the actual production code.

They perform this activity *after the fact*, so that what is in the release folders trails the running system. This approach aims merely to capture and archive what's already in production, whatever that happens to be. True SCM, on the other hand, tags and selects in advance the components for building a new version of the application. The release is then built using this information.

This difference is crucial. In the *capture-and-archive model*, the application in production use is the only embodiment of the product's true state and the SCM process follows along, trying to keep track of things and provide developers a reliable baseline to build on. In a true *baseline-and-release model*, the release engineer creates the baseline a priori, using the SCM system to build a new version of the application from previously identified configuration items, and then releases that version, exactly matching the baseline.

Organizations that use a slipstream development-and-delivery process usually rely on the capture-and-archive baselining model. These organizations do not offer true software product releases, per se, just a continuous stream of updates—or patches—to the software in production use. Although this may be convenient for emergency fixes, if overgeneralized it can become the main way every change enters production.

When an organization slips into slipstream development and delivery, it never replaces the entire application. Instead, it updates the application incrementally, which creates some complex problems:

- The continuous-stream model makes it difficult or impossible to recreate past instantiations. There is no way to compare the application as it runs now to the application as it ran several updates back, nor is there a straightforward way to roll back to a previous update stage.
- Testing the evanescent incarnations of the continuous stream can be difficult or impossible—and very expensive. There is rarely a discipline in place to test the full application with each individual changed file—and only that file—and to refuse additional changes until all bugs found in that piece have been resolved and the application has been successfully regression tested with just that changed file.
- The continuous-stream model can also present a challenging security problem. Once a piece of code makes it into production, it won't be replaced until the next fix to that file enters production. If someone slips in a special copy of a piece of code, that piece could stay in production for a long time with no one ever noticing. Although this sort of thing can happen with the replace-the-application model, at least in that case such alterations should leave some tracks in the source code.
- Over time, the always-update, never-replace aspect of the continuous-stream model can lead to the suspicion that some pieces in production weren't built from the source code in the development's current release source folder, such as a live hot fix that didn't make it back into the source folder. This leads to the fear that a full rebuild from the source will not duplicate the running system. If you can't re-create the running system, you're not in control.
- Having reached this uncomfortable position, the development team elects to rebaseline their source code by capturing and archiving the system in production. There is no branching model. This approach simply replaces one line of code with another, at which point the stream starts anew.

The continuous-stream model also implicitly assumes that developers can manage hundreds of files into production well enough, one at a time. Down that path lies madness. Even if the individual files moved along in a completely automated, foolproof process, it is still necessary to manage, test, and track hundreds of interdependent parts as they progress toward production. Even if only 10 percent of the files remain in play between each capture-and-archive baseline event, that's still an impractical number of components to juggle.

releases as required. Further, it avoids the problems caused by branch-by-release as exemplified in the build-by-bug-number syndrome. The branch-by-purpose model will satisfy all typical evaluation criteria.

This model offers the additional benefit of simplifying things for developers. It lets them work in the same environment, the main dev branch. This reduces confusion among the development team members about where to make changes, makes for more robust emergency releases, and reduces the team's angst about code chill and code freeze.

We have repeatedly seen reasonably mature and disciplined development teams heatedly resist code freeze in the branch-by-release model because it left them no place to check in their in-progress work. Delaying code chill or code freeze usually compresses the testing cycle, lowering release product quality. Again, forking a new release branch instead of a new ongoing work branch just works better.

On the other hand, managing this model is more complex, primarily because it requires a more sophisticated understanding of SCM and a more sophisticated use of SCM tools. Further, it turns the

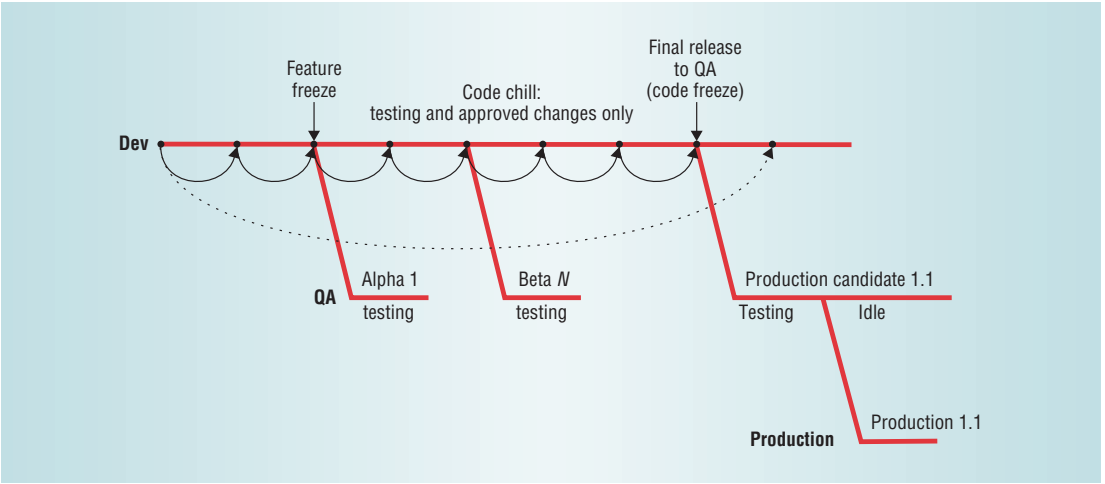

Figure 4. Avoiding the build-by-bug-number tar pit. In the branch-by-purpose model, the developers can just check the code back into the main development line after the release's code freeze, leaving the release code untouched and its integrity intact.

conventional approach on its ear: As before, the release engineer spins off a new branch for the product being released, but development remains on the main dev line.

The product may be released to QA for testing, in which case the release engineer designates a QA branch. Or, once the developers and testers have completed the defect find-and-fix activities, the release engineer can release the product for production use—directly to the user in the case of contracted software, to an operations group in the case of in-house software, or—in the case of shrink-wrapped software—to the manufacturing entity. When releasing the new version, the release engineer creates a production line branch—a special branch whose sole purpose is to support the released version.

The branch-by-purpose model presupposes that the product release cycle includes a *feature freeze* milestone after which developers add no further features and make no more enhancements to existing features without careful change evaluation and control. This milestone marks the entry of development activity into *code chill*. During code chill, developers make fixes in the dev line and periodically send new releases containing the fixes to QA. Each release to QA has it own branch, which makes it possible to verify whether any given bug appeared in an earlier release to QA, and to identify both the release in which a bug is found and the release in which it is fixed.

As the software gains stability through the testing cycle, fewer and fewer fixes become necessary, until code chill gives way to *code freeze*. At the code freeze milestone, the team presumes the product ready for final production use, even though it is only a production candidate that must still undergo final testing to assure its production readiness.

From this point on, the developer applies any fix approved by the change control board (or the project management team) directly to that QA branch and migrates the fix to the main dev line. After testers have verified that the changed code base works properly, release engineering can release the product to production and create a production branch. The team uses this same sort of cycle when a bug found in the field requires an emergency release, as shown in the Production 1.1 build in Figure 3.

SUPPORTING PARALLEL DEVELOPMENT

Today's prevailing climate of rapid development demands parallel development. The software project must proceed with enhancements not intended for the release currently in progress. Even in those rare cases in which rapid development is not de rigueur, the software project may have staff available during code chill and code freeze to implement bug fixes for future releases.

To satisfy either of these needs for parallel development, release engineering can create temporary bridge lines, one for those pending bug fixes and one for the enhancements, or just one bridge line for both bug fixes and enhancements. Then they can merge that line back down into the main dev line after the production candidate releases.

In the branch-by-release model, on the other hand, a problem arises when an in-progress code change must be delayed until after a release. In the case of orphaned check-outs that didn't make it into the release, developers find themselves stuck between a rock and a hard place because they must

- either check the changed code back into the line it came from, which means putting it in the branch with the released product that didn't contain it, thereby destroying the integrity of that branch, and then migrate it to the new release branch; or
- check it out all over again from the new branch, redo the changes, and check them back into the new branch.

The branch-by-purpose model obviates this dilemma. As Figure 4 shows, because the main development code line hasn't changed, the developers can just check the code back into the main

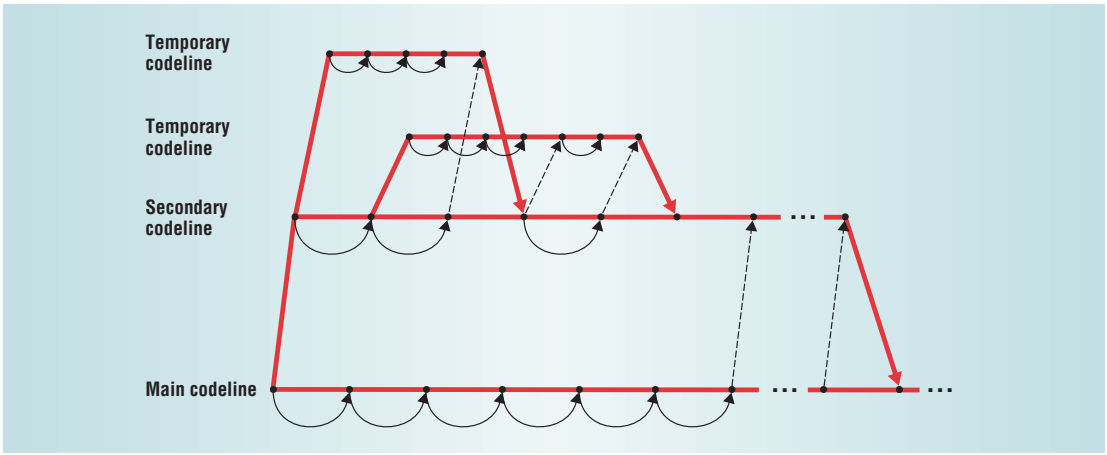

Figure 5. Temporary codeline changes. The release engineer establishes a secondary codeline to handle a major product change. Developers can then branch off one or more temporary codelines from this secondary line and use them to work out minor changes.

line from which it came, and it will be included in a future release's product release cycle. The code base for the released product, Release 1.1, remains untouched because it resides on its own branch.

Multiple groups and subprojects

Branch-by-purpose more readily accommodates the need to have multiple groups working on multiple subprojects in parallel. Branching can establish a fully replicated environment in which developers can modify code and test changes without impeding other developments occurring simultaneously in the code base. Creating small, purposeful, alternate development lines lets developers check their code into the appropriate temporary line without affecting the main line.

The parallel development concept recognizes the need to frequently merge changes from one line into another. Because the branch-by-purpose model anticipates that need, the SCM and release engineering processes can manage merges by design, making them as painless as possible.

In contrast, the branch-by-release model operates from the underlying assumption that releases are linear and sequential, each subsequent release flowing immediately from its predecessor. Thus, changes made to an existing release of the current baseline—or its predecessor, if still in the field—are somewhat unnatural, and making these changes requires special effort to ensure that

- any re-release of an old release contains just the desired changes—typically, bug fixes—and no others, and
- the release engineer properly merges these changes into the new, in-progress, release and into all releases between the old release and the current release.

In the branch-by-purpose model, on the other hand, developers work from the underlying assumption that they are creating a single main line of code and other artifacts and that, at predetermined times, release engineering will create a separate branch when it releases a new version of the product. This approach enables frequent builds from the top on the main line of code that thus include all available changes. The best practices of today's software companies emphasize frequent integrations via builds from the top. Such builds integrate changes bit by bit, as testers and developers check them in. Builds can be nightly or continuous. This practice assures continual integration and avoids the big bang that lots of colliding changes can cause when the project team delays integration until the release cycle's end.

Having a fully replicated environment for modifying code enables continuous or nightly builds of each line and further testing of those changes, without impeding the main line's development. It also lets multiple developers make multiple changes to the code for multiple purposes.

While project teams can perform branching for parallel development at very fine granularity levels—down to each bug fix—they use two common modes to create branches for alternate development lines:

- when one or several developers will be working on a small-to-middle-sized change or related group of changes over several weeks, or
- when many developers must work on a large change over several months.

Variations of the branch-by-purpose model can accommodate each mode.

Codeline changes

When testing shows that changes work as expected, the developers merge the changes into the main development line. As Figure 5 shows, developers also periodically merge main codeline bug fixes and other changes to the temporary line if it must be maintained for more than a few weeks.

Whether or not developers periodically merge the main line's changes, they must merge the changes up to the temporary line before merging it

back down to the main line. Integration takes place on the temporary line, assuring that its changes are compatible before inserting them into the main line, which the rest of development depends on.

To make larger, more complex changes, developers establish a secondary codeline to accommodate a major product change, such as a rearchitecting effort. They merge the changes to the main codeline up to their secondary line every week or so.

Figure 5 shows two temporary lines branching off the secondary line to let developers work out the kinks in some small- or medium-sized changes that depend on changes in the secondary line.

Despite its wide use and its appearance of mirroring the conventional standard of sequential baselines, the branch-by-release model imposes unnecessary burdens on the software project team in supporting released versions of software while developing new releases. Error-prone, it also often requires manual handling of fixes to released software. Further, its fundamental flaws include unnecessary complexity in managing post-release code fixes and unnecessarily orphaning changes in progress, which cannot be successfully injected prior to a scheduled release date.

The branch-by-purpose model, on the other hand, avoids these pitfalls. In addition, it provides a structured mechanism for managing multiple lines of code, supporting the organization's need to deliver multiple releases as well as development's need to shorten cycle time by using parallel development and continuous integration. ■

Acknowledgments

We thank Matt Foley and Adonica Gieger for their helpful comments in reviewing this article.

References

1. P.H. Feiler, *Configuration Management Models in Commercial Environments*, tech. report CMU/SEI-91-TR, Software Eng. Inst., Carnegie Mellon University, Pittsburgh, 1991.
2. *ANSI/IEEE Std. 828-1983, IEEE Standard for Software Configuration Management Plan*, IEEE Press, Piscataway, N.J., 1983.
3. *ANSI/IEEE Std. 1042-1987, IEEE Guide to Software Configuration Management Content*, IEEE Press, Piscataway, N.J., 1987.
4. R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, no. 2, 1998, p. 233.
5. I. Sommerville, *Software Engineering*, Addison-Wesley, Reading, Mass., 1996, p. 657.

Chuck Walrad *is managing director of Davenport Consulting. Her research interests include strategies for the propagation and practical implementation of best practices in software engineering. Walrad received an MS in theoretical linguistics from the University of California, San Diego. She is a member of the IEEE and the ACM. Contact her at cwalrad@daven.com.*

Darrel Strom *is a senior software engineer at Expert Support. His research interests include developing software factories that automate the critical processes required to build and deliver software products reliably. Strom received a BS in computer science from the University of Southern Mississippi. He is a member of the IEEE and the ACM. Contact him at dstrom@xs.com.*