End-of-release Branching Strategies

Part 2: Release Branches

By Steve Berczuk

Read End-of-release Branching Strategies, Part 1

In part one, I discussed the active development line approach to development, in which you defer branching and develop strategies for maintaining working code. While branches carry overhead, branching at the end of a release can provide benefits.

End-of-release Branching

Creating a branch at the end of a release can give your team flexibility. It provides a place to do final integration testing and fixing, while allowing work to start on the next release. And, once you ship, it is easier to provide an urgent fix to code being used by customers.

If your team is able to deliver frequently enough to address problems in an iteration and if your architecture allows you to hide incomplete or unwanted features from customers of a current release, you may be able to deliver fixes on the active development line rather than a release branch. Most teams are not at that point of maturity and need a mechanism to work on new features while still being able to address problems with a prior release. Even for teams that embrace the agile ideal of continuously shippable code but have not yet attained it, the release line pattern is valuable. Without the isolation of a release branch for code done "the old way," progress would be difficult. As time goes on, a team's engineering practices improve, and the ideal of always-shippable code becomes more reachable.

The challenge is deciding when to create a release branch.

End-of-project Branches

Once a project is complete, the team often creates a release branch. If all goes well, this branch need not be touched, and the team can release their next version from the main line.

In reality, boundaries aren't always that clear. For example, your team may want to do more extensive integration testing on a "feature complete" product. You may not expect to find problems, but you need to be prepared to change gears and fix any problems that arise. Or, your team may believe that the product is done but is waiting on feedback from a key stakeholder before you can ship the code. While you are waiting for this feedback, you want to start work on new features. Needing a period of time after features are "complete" to stabilize a release isn't an ideal situation, but it is a common one, and configuration management practices can help you manage the process of stabilization.

There are three common approaches for deciding when and how to branch:

- Create the release branch when you are "feature complete," and plan to fix last-minute issues on this code line. In this case, the release branch is really a "release-prep code line," as described in *Software Configuration Management Patterns*, since you expect that there is still work to be done.
- Change your work style to avoid final integration work, working off of the active development line.
- Branch for the new work by creating a task branch and merging that work into the active development line after the release is complete.

Let's consider each option.

Branch Early: Release-prep Branch

Creating a branch when you are feature complete frees the main line for ongoing development and provides you with a ready-made

release line. If the number of problems you expect to see is small and isolated, the majority of the team can work on the main line with only a few people working on the branch on an as-needed basis. (Part of this equation is how able team members are to work across components). A *release-prep branch* is preferable to a code freeze, which would leave team members idle or, worse, working on code outside of the source-management system.

A release-prep branch would have these policies:

- Only make changes that are critical to shipping the product.
- Review changes and test them a bit more strictly than changes to main-line code. This is to avoid regressions that could delay release.
- Integrate fixes for problems discovered on the branch to the main line to avoid regressions in a future release.

This approach is a good compromise between being ready to ship at an iteration end and keeping the team deadlocked until the last of the issues in a release are ironed out. If the time between branch creation and shipping is small, the process can work well with few downsides.

When the time between branch creation and product release is too long, merging code between branches can add a drag factor to the work. You can mitigate this risk by having the person who did the original work merge changes, as he understands the context of the code and has the ability to understand conflicts. Merging (and deciding what to merge) takes time, adding to the cost of both the release work and any work going on in the main line. Also, as time progresses code lines may diverge, causing additional problems.

Soon after a branch is created, the code is similar enough that mechanical merging using diff tools can work well. The code lines can diverge rather quickly, especially on an agile team that is practicing refactoring, and merging increases in complexity, changing from "moving code" to "integrating functionality." A fix to code on the branch could now involve a file that no longer exists on the main line. You need to evaluate each issue discovered on the branch to determine whether it applies to the main line and, if so, where to apply the fix. In the best case, you manage to evaluate all of the changes and apply them as appropriate. In the worst case, due to the ineffectiveness of mechanical merging, you may discover a number of un-merged changes that will hamper the effectiveness of your SCM tool's merging capability.

Even though a release prep branch is preferable to a code freeze, branching before the code line is in a state where it can be released soon can introduce schedule risk to the team.

Sidebar: Who Merges

In some organizations, there is a dedicated team of people responsible for merging changes between code lines at a regular frequency (e.g., weekly). This means that the person doing the merge is missing the development context of the source code and so cannot easily resolve merges. At best, this means that merges take longer. At worst, the merges can result in problems that are detected by those working on the target code line. If possible, the person working on a code line should do the merge. The person doing a change on the branch should merge changes into the trunk, perhaps consulting with someone who is working on the trunk. Those working on a task branch should merge changes from the main line. And, everyone should merge more frequently.

Branch Late: Active Development Line

Another approach is to go to the other extreme—not branching until your code is complete. If you are an agile team, you want to focus your efforts on the highest-value activity, which often is releasing the current product. A common belief at this point in a project is that that the stabilization work can be done primarily by a subset of the team, and the obvious solution is to branch as soon as the feature-development work is complete. It might make more sense to focus the efforts of everyone on the team on getting the code ready for release and working on the main code line—the *active development line*. This approach may involve people on the team working in areas in which they don't have lots of experience. But, if your team is composed of generalizing specialists who can work beyond their nominal areas of expertise, your team can increase its throughput, as everyone on the team can contribute and you are not limited by the capacity of the one person who knows the code where a problem is.

While this single-code-line approach has the advantage of simplicity, it may not make practical sense if your organization does not have the engineering practices in place to support it. Errors made in the main line can further delay release, and practices such as testing to avoid regressions are key to allow for a readily shippable main-line product. And, you may still find that the end of the release will involve more waiting on feedback than coding, so your team can be underutilized if the organization around the team is not able to give feedback on fixes in a timely fashion.

Branching in Reverse: The Task Branch

A third approach—doing the next release work on a *task branch*–keeps the primary focus of work on the main line while still allowing for parallel work. A task branch is a branch where a group of developers can work on a task or feature in parallel with the main-line work. A task branch has the following policies:

- The task branch should integrate changes from the main line frequently.
- Developers on the task branch are responsible for merging appropriate changes from the main line.

The advantage of this approach is that those working on the soon-to-be-released code are not delayed by merging changes, though

they may need to be consulted when the task branch developers are merging changes. This will speed up the work on the released code. Also, the most immediate work is being done on the main line.

Once the main line seems stable enough to ship, the team can create a release branch immediately or a short-lived release-prep branch.

Summary Guidelines for End-of-release Branching
A rationale for branching is to isolate code at the end of a release so that it can stabilize. Isolation through branching often masks a quality problem that will end up manifesting itself in the added cost of maintaining parallel streams before a product is released. Branching is easy. Rather, it's the merging and the cognitive overhead of understanding how changes flow between branches that are difficult, so it's important to choose a process that minimizes the cost of branching and merging. Here are some guidelines to follow:

- Use fewer code lines to keep things simpler, better, and lower cost.
- Branch as late as possible. Create a release branch when the code is in shippable form.
- Keep the highest-priority work on the main line.
- Keep the bulk of the team working on the main line
- Regardless of the branching model, integrate changes from the to-be-released code to the next-release code frequently (e.g., hourly) to avoid painful integrations later on.

While the best approach to managing the end of a release is to develop a testing strategy that makes it easier to keep your code shippable throughout the development process, some teams may need to stabilize feature-complete code before shipping. Before branching by habit to create a stabilization branch, consider the cost of branching and the value of keeping the most important work —your next release—on the main line. You can still work in parallel and get your releases out the door more quickly by helping keep the team's energy focused on delivering quality code, not branch management.

Further Reading
Software Configuration Management Patterns, by Stephen P. Berczuk and Brad Appleton.
Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, by Jez Humble and David Farley.

Read End-of-release Branching Strategies, Part 1

About the Author
Steve Berczuk is an engineer and ScrumMaster at Humedica where he's helping to build next-generation SaaS-based clinical informatics applications. The author of Software Configuration Management Patterns: Effective Teamwork, Practical Integration, he is a recognized expert in software configuration management and agile software development. Steve is passionate about helping teams work effectively to produce quality software. He has an M.S. in operations research from Stanford University and an S.B. in Electrical Engineering from MIT, and is a certified, practicing ScrumMaster. Contact Steve at steve@berczuk.com or visit berczuk.com and follow his blog at steveberczuk.blogspot.com.

StickyMinds.com Weekly Column From 12/13/2010