JOIN OVER 100,000 PROS! Join Log In | Member Benefits

STICKYMINDS

Agile Testing DevOps Security AI

An Agile Perspective on Branching and Merging

ARTICLE

By Steve Berczuk , Brad Appleton Robert Cowham | November 30, 2005

SUMMARY

This article focuses on branching and merging. We present some background for branching and merging, and consider some of the implications for agile development in particular. We also hope to reduce some of the suspicion that many agile developers have of branching. The article assumes some overall branching knowledge and yet revisits some particular details that often seem to confuse people. This is a fertile area which we will continue to expand on in future articles.

Branching and Merging

Branching is a mechanism for isolation and parallel development which is useful in many situations as discussed below. The advantage of branching *with good tool support* is that it allows you to merge changes between branches more easily, and in a less error prone manner than manually making the same code changes to the two separate branches (see **Introduction to Branching** from Streamed Lines).

There is little lasting value in branching if the branches will diverge indefinitely without propagating any changes. It may be seen as a short-term win that allows a new project to get up and running very quickly from an existing codebase for another product. But the gains of such persistent variants are very short-term, and the resulting long-term costs of proliferating code and effort across the same codebase can quickly overwhelm whatever short-term gain is to be had. So whenever we branch we almost always will need to merge back to another branch. Merging changes between branches does not come for free, and the associated costs and implications need to be carefully considered when deciding when and how to branch.

Agile Branching and Merging Considerations

Early and frequent feedback is a key element in all agile methods. Practices such as Continuous Integration help ensure integration issues are never left to fester for a long periods of time and then come back and bite you at inopportune moments (just when you are trying to get a release out the door), but instead confronted and resolved on a

We use cookies to ensure you get the best experience on our website. Accept

Read More

for some scenarios where branching is usually a better fit, such as maintaining multiple releases. Of course, branching is not a panacea but the need to branch often stems from business reasons rather than technical ones. Maintaining multiple releases or custom variants is typically a business decision arising from a contractual agreement, or Service-Level Agreement (SLA), that is often a source of considerable additional revenue. So even though multiple maintenance may come at additional cost, it may often be fed by an additional revenue stream for that specific purpose. Branching is quite often the best way to meet those needs, and there are ways to branch that still allow us to be "agile" and focus upon delivering business value to our customers.

We have seen a number of projects work for considerable periods of time (months if not years) using a single mainline with no branches. The fact that they could is a testament to the soundness of their practices. Yet we as SCM professionals regularly encounter misunderstandings of what version branching and merging is all about, specifically what the state-of-the-art is as regards tools and processes.

Automatic Merging
The state-of-the-art in merging tools has come a long way. However, it bears repeating that there is no such thing as a 100% reliable merge program. Human intervention is invariably required at some point (almost reassuring in that there will still be some jobs around for the foreseeable future). All automated merge programs have their strengths and weaknesses but most are performing merges at the level of blocks of text. There is usually no difference in the algorithm for used for merging two text files containing poetry and two text files containing C++/Java or any other language you care to mention. Indeed, once it is possible to perform 100% correct merges at the semantic and syntactic level, the merge system may well be able to write whatever program we need 5 minutes before we know we needed it!

Having said that, we believe that the progress made in IDEs which support refactoring for some languages will support merging at the level of the parsed language tree rather than just purely at the textual level (at least for languages such as Java). For example, renaming a variable is not significant regarding the meaning of the resulting code, but can mean many different individual lines have changed between two versions. If one person renames the variable and another edits lines which use the old name, then the real changes will be obscured by "conflicts."**Some Detailed Merging Examples**
For clarification, let us look at some detailed examples of types of merge - a couple of examples clarifies pages of theory!

Laura Wingerd presented at a recent conference on the flow of change (a chapter in her **new book from O'Reilly**). She introduces the concept of the "Tofu Scale" to describe the properties of codelines and how firm or soft they are! If you draw firmer codelines above softer ones (in the diagrams below the mainline is firmer than the personal branch), then catch-up becomes "merge down" and publish becomes "copy up."

We use cookies to ensure you get the best experience on our website.

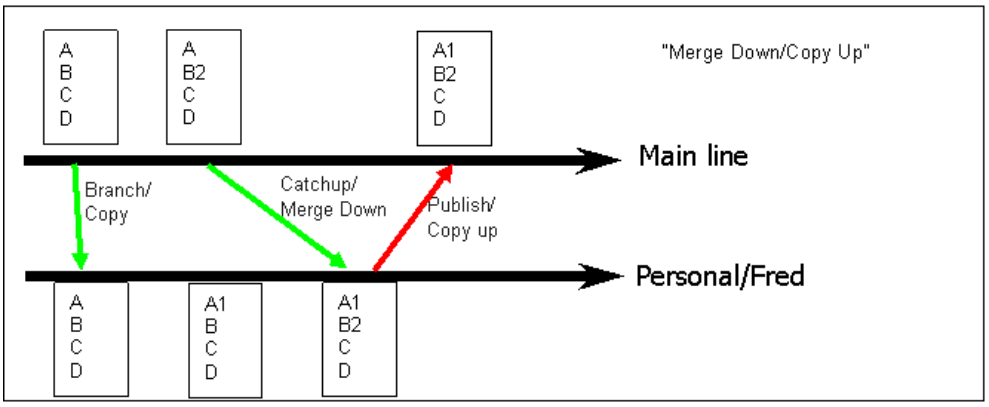


Figure 1 - Simple Use of Personal Branch with Merge Down/Copy Up paradigm.

In figure 1, the file starts out with contents [A,B,C,D] (where each letter represents a block which may be 1 line, 10 lines or any number of sequential lines in the file) and is branched to the personal branch which is just a copy. On the mainline, B is changed to B2, and on the personal line A is changed to A1. Performing a "merge down" gives [A1,B2,C,D] (assuming for now that the two changes are compatible). The merge down performs the (relatively) risky merge in the codeline which is "softest" on the Tofu scale! In this case, the merge is not that risky. The publish operation is simply a "copy up" of the resulting file back to the Mainline, which now includes the change.

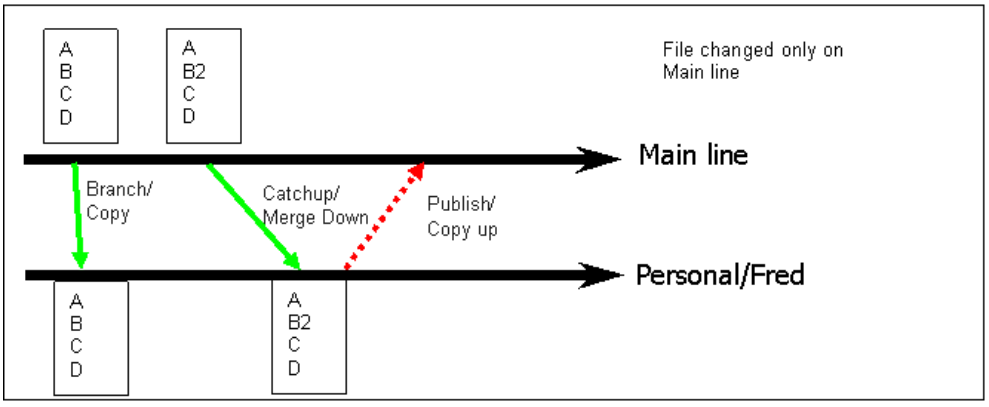
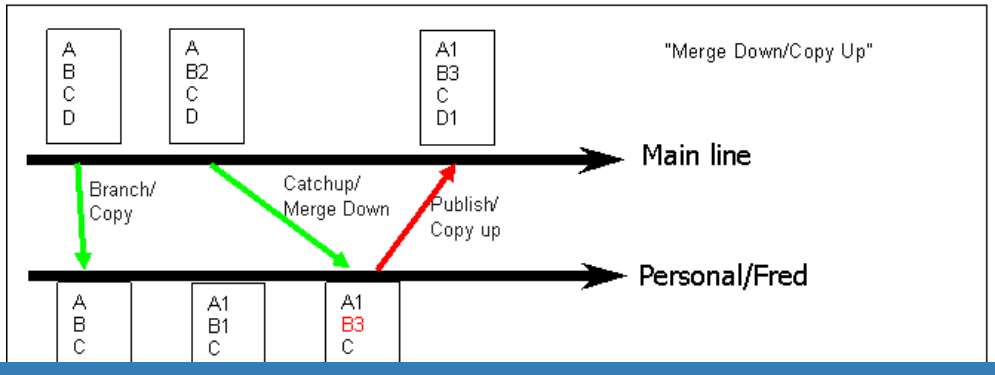


Figure 2 - File not changed in Personal branch - change brought in from Mainline

Figure 2 shows a file that is changed only in the Mainline and brought into the personal Task branch so that it can be tested together with all other changes from the Mainline, including those that required merging. Note that the publish step is not required, since in this case there is not further change to go back.



We use cookies to ensure you get the best experience on our website.

In figure 3, we throw in a conflicting change such that B1 and B2 clash, needing to be manually edited and turned into B3. This sort of merge is more risky, and would require to be built and tested locally (*Private System Build* and *Smoke Test)* before checking in. However, the publish is the same as before - just a copy from source to target.

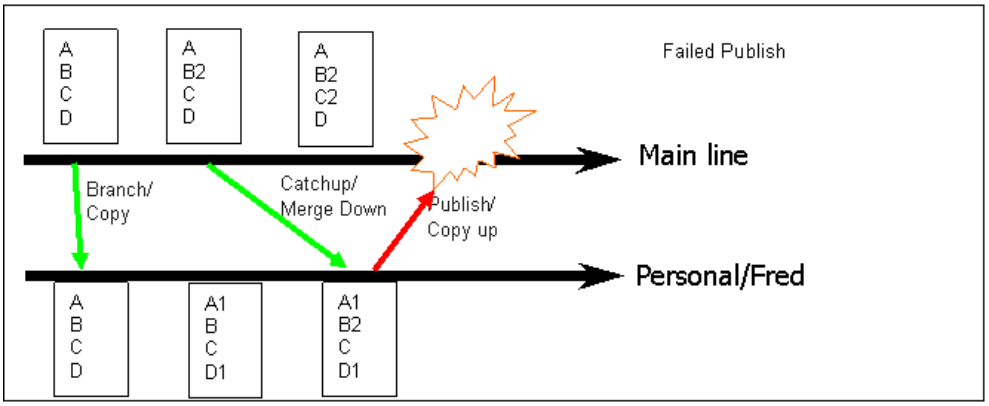


Figure 4 - A Failed Publish

In the first 3 examples, the merge into the personal branch is done using normal merge facilities and may require checking and certainly local build/text before checking it in. However, in all examples, the publish is just a copy, which should be fully scriptable in any reasonable SCM tool (including some straightforward validation to ensure that it really is a copy and that nothing has changed on the mainline since the last catch-up). Thus the script must catch the example in Figure 4 where there is still a Catch-up needing to be done, and thus the Publish is rejected.**More on the Merging Problem** Working in a *Private Workspace* may result in conflicts which need resolving. With more than one person doing development, and even in some cases with single developers on a project, merging is required on a regular basis, particularly when you perform continuous integration: Two people both checkout the same version of the same file; They both make changes; and one person checks in first. Now the second person has to merge their changes together with the first person's.

This is the same problem as merging between two branches. The difference is the length of time of the "divergence" gradually increases the set of divergent files between the two codelines and hence increases the likelihood of concurrent edits (and the resulting need for a merge).

So, avoiding branching does NOT eliminate the merging problem. It only somewhat decreases the likelihood. What branching decreases more-than-slightly is the resulting complexity of the merge when it does happen. The more activity that has taken place in parallel on the same sets of files, the more complicated it may be to merge those files. If both codelines are very active—this will occur a lot. Most of the time this is not the case. Instead, only one of the codelines is very active (the one that works on the "latest stuff") and the other codeline is for the occasional repair that has to happen in the legacy version (and propagate forward).

The same holds true for most workarounds people come up with to avoid branching (for example conditional code without version-branching). If you don't branch and handle it with run-time code, you have to have separate run-time configuration settings for each

We use cookies to ensure you get the best
experience on our website.

If instead we branch, we may have to do two builds, but we might execute the exact same sequence of steps (procedures) to do the build in either codeline because we will typically have "branched" the build tools and associated configuration files too. So the procedure to build either codeline would in fact be less complex than the one needed to avoid branching, because that "defers" the configuration-decision from checkout-time (on a new branch) to compile-time instead (for a different set of compilation flags). Therefore, we don't feel that avoiding branching avoids complexity. It's a question of choosing which part of the rug to "sweep the complexity" under. If you fear branching (or merging) and choose to never sweep it under the earlier portion of the rug, that is the choice you make, but the complexity still rears its ugly head in some form or another. The key is to recognize the choices and avoid being afraid of branching and learn to use it wisely—taming the branching beast. This process of education is part of our mission as agile SCM-istas!

The Refactoring Blues

Having said the above, one major complication for branching and merging, and yet also paradoxically a reason *for* branching, is **refactoring**. Agile teams want to refactor and indeed need to be able to do so to preserve a clean codebase which is more easily maintainable in the future. Simple refactoring does not change the functionality of the code but may change the textual format substantially. It can make it very difficult to promote a maintenance change (bug fix or perhaps enhancement) to a class in a newer version when that new class has been totally rewritten (because of refactoring) in the new version.

Refactoring becomes even more problematic when dealing with files that are renamed. This has particularly come to the fore with the rise of Java, where the class name must be the same as the name of the file in the file system (possibly a questionable design decision, but one we now have to live with).

Some IDEs, starting with IntelliJ and now including Eclipse and others, offer a number of refactorings from within the IDE, including the renaming of classes. This can result in a number of changes throughout the application where that class is explicitly referenced in other classes—all handled by the IDE, including automatic checkouts and renames in the underlying SCM tool if that tool's plug-in supports it. Handling renames across branches is one of those areas where there are different levels of SCM tool support—make sure you are using one that supports it well—or at least well enough.

The problem becomes worse if you have serious parallel development going on. If 3.5 is your base release and you have two major enhancements A & B in development. The business doesn't know if A will turn into 3.6 and B into 3.7, or the other way around. Or perhaps both will merge together into 3.6. Until the modules (which share a common database and rely on some common classes) are close to complete and have a release cycle planned concretely it's hard to refactor any of the base classes because then you make dependencies about them both being released together, or one before the other.

Note that the results of this type of requirement and the associated overheads need to be made visible to management. They often don't realize directly the maintenance

We use cookies to ensure you get the best
experience on our website.

have been doing refactoring in parallel development projects for more than a decade, and this would not have been possible without branching (and good tool support and knowledge of how to use those tools). Using strict code ownership and file-locking would have made things too unbearably slow to be productive. With sound parallel development practices that group together modifications made toward the same purpose as logical (rather than physical) units of change, the projects had little or no hassle and no waiting around for other folks to check something back in to release a file-lock.

This highlights the fact that the most successful parallel development shops don't do parallelism at the granularity of single files or classes/methods. They do it at the level of projects and subprojects and tasks. If you do it at the file/class level granularity it becomes too complex track. If you do it at the project/task granularity it all fits into a single change-flow hierarchy. The "Streamed Lines" paper is clear about the requirement for project-oriented versus file-oriented locking. Otherwise it looks as if all those practices are referring to individual files and it seems crazy. This also raises the need for a version control tool that can not only do branching, but that lets you use symbolic names for branches (not just revisions on the branch, but for the branch itself). If the tool tracks both divergence (branch-points) and convergence (merge-points) then merging is no longer the nightmare that everyone always assumes it must be. In fact it's often quite trivial. And some of the more modern tools even provide some automated and intelligent merging support to make it a breeze most of the time.

Case Study—Changing Business Requirement Forces Branching

One example was an organization producing a Java/HTML based web-service, hosted by themselves. They were a pure XP shop, and had been working with a single mainline for over 2 years, making fortnightly releases. Very few bugs had escaped through to live production, and the few that did could usually wait for the next release, less than 2 weeks away.

However, business requirements changed, and in this case it was functionality that the business decided to implement and get ready to go live, and yet was lacking the final signature from the client to allow it to be released. With just a mainline, these changes either had to be backed out or they held up the whole release—neither situation a very pleasant thought. This scenario was not just a one-off either—so they introduced a release line and started to "promote" code to it that was fully tested and authorized without affecting the implemented but not fully authorized changes. This worked very well for them.

Case Study—Branch on Conflict

In their paper presented at the BCS CMSG 2005 Conference, **SCM in a Large-scale Agile Development**, James Dyson and Paul Spalding describe a problem they encountered where functionality needed to be removed "at the last minute." Similar to the previous case study, "there were occasions where, towards the end of an external iteration, things had changed and the business either wanted to remove some completed functionality, or wanted to de-prioritize some incomplete functionality in favor of spending more time on some newly-important requirements." The tool they chose, CVS, did not support this partly because of the lack of atomic checkins. Other tools do support this more readily than perhaps they discovered at the time (which did start in 2002 when such tools were fewer on the ground).

We use cookies to ensure you get the best experience on our website.

is in the workspace this is a consistent working version (passes all unit tests). This is similar to the *Private Checkpoint* pattern mentioned in our previous article. In this instance, an automatic *Private Branch* would be better *Private Checkpoint* since the conflict needs to be recorded for posterity to help the removal of the functionality if the business wanted it.

This sort of functionality is not difficult to do in those tools supporting atomic change-sets and good branching with some fairly simple scripting if necessary as shown below.

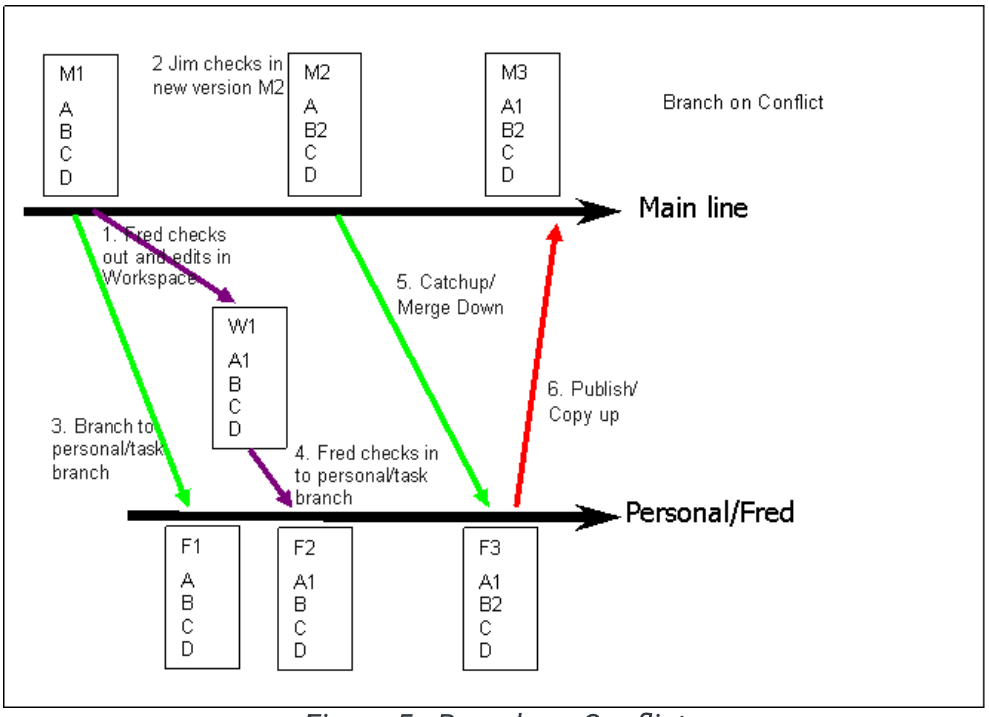


Figure 5 - Branch on Conflict

Figure 5 shows the sequence of events for a single file (M*n* is the name/revision on the Mainline, F*n* on the Task branch). Of course behind the scenes the script would need to branch all the files in the change-set Fred was working on so as to have a consistent change-set in the Task branch. The naming of the Task branch could be created automatically using some simple heuristics, possibly including the date/time of the change or some similar identifier. While this might at first sight appear complicated it is not difficult to do and the results are the storing of the stable implemented function which would be very easy to remove cleanly if future business needs dictated.

Conclusion—Making Branching More Agile

As has been know to SCM practitioners for a long time, branching and merging are very capable tools to have in your toolbox, and yet we find that many developers regard them with suspicion. This suspicion has carried over into agile circles, and most of it is unwarranted! Branching *can* be used in an agile way to provide a useful level of isolation and security without losing any speed.

This does require that you become both comfortable with branching theory, and in particular comfortable with how your tool does it. If in doubt get some training—it can save you days if not weeks of work! In the future, we will put some more theory around

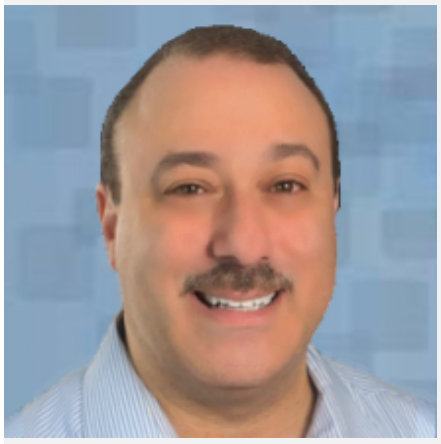
We use cookies to ensure you get the best experience on our website.

Topics: agile,

ABOUT THE AUTHOR

Steve Berczuk

Steve Berczuk is a Principal Software Engineer with experience as a manager, Scrum Master and technical lead in Boston, MA. 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 ScrumMaster. Contact Steve at steve@berczuk.com or visit berczuk.com and follow his blog at blog.berczuk.com.



Brad Appleton

Brad Appleton is an Enterprise Agile+DevOps leader, coach & manager, and seasoned DevOps/ALM/CM solution architect at a large Fortune 100 company. Currently he helps organizations and teams scale, adopt and apply lean/agile development methods and DevOps/ALM/CM practices and tools. He is co-author of *Software Configuration Management Patterns*, the Streamed-Lines branching patterns, and a former section editor for *The C++ Report*. You can read Brad's blog at blog.bradapp.net.

Robert Cowham

Robert Cowham has long been interested in software configuration management while retaining the attitude of a generalist with experience and skills in many aspects of software development. A regular presenter at conferences, he authored the Agile SCM column within the CM Journal together with Brad Appleton and Steve Berczuk. His day job is as Services Director for Square Mile Systems whose main focus is on skills and techniques for infrastructure configuration management and DCIM (Data Center Infrastructure Management) - applying configuration management principles to hardware documentation and implementation as well as mapping ITIL services to the underlying layers.

We use cookies to ensure you get the best experience on our website.

LETS HANG!

FEATURED RESOURCES

SECURITY

Shifting Security Left: DevSecOps eGuide

White Paper | StickyMinds

Log In or Register to post comments

USER COMMENTS

3 comments

Kylie Wilson | January 28, 2014 - 6:40am EST

Very insightful article. I came across www.pmstudy.com which has some great free resources. They also have a free test which can gauge your readiness to take the pmp exam.

Log In or Register to post comments

We use cookies to ensure you get the best experience on our website.

Log In or Register to post comments

Helena Lui | March 26, 2016 - 1:11am EDT
very insightful and well researched article. great job! you can check out www.examspm.com for free PMP training resources.

Log In or Register to post comments

YOU MAY ALSO LIKE

SECURITY
Agentic AI Opens Up New Security Flaws
Nahla Davies | August 27, 2025

SECURITY
Countering the Threat of Ransomware-as-a-Service (RaaS)
Nahla Davies | August 22, 2025

SECURITY
Building Trust in Blockchain: The Role of Smart Contract Testing
Ekaterina Egorova | November 18, 2024

Join the crew for FREE! Access exclusive content, contribute, and more.

See All Member Benefits

We use cookies to ensure you get the best experience on our website.

We use cookies to ensure you get the best experience on our website.