Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Three-Way Merging: A Look Under the Hood

Automating three-way code merges requires considerable sophistication from the version control system.

December 24, 2013
URL:http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

For several years, Pablo was a blogger for Dr. Dobb's covering Agile technologies and best practices. With this article, he returns as a regular blogger. His next post will appear in the left panel of our home page, along with our regular bloggers. —Ed.

When I run a demo of our source code merge program, a developer occasionally raises a hand and asks, "What do you mean by 'three versions' of the file being merged?" To answer this and explain how three-way merges work and why they are important, let's start by taking a look at the traditional two-way merge.

What Is a Two-Way Merge?

Suppose we're modifying the same file concurrently. You go and make some changes to the file and then I make some more changes.

At some point in time, someone looks at the two copies of the file and they see something like Figure 1:

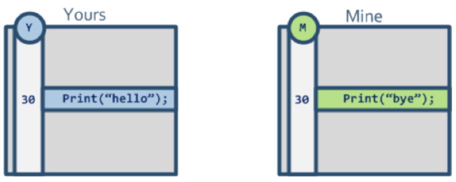


Figure 1.

This third person looking at the files sees there's a difference on line 30 but:

- How can he tell whether you modified line 30 or if I modified it?
- What if we both modified the line? How can he tell?

He can't.

He will have to call both of us and trust our memories to find out who modified what. And yes, fortunately we programmers never forget, right? ;-)

This "third person" is likely to actually be the version control system trying to do a simple two-way merge, which just compares two versions of a file and tries to merge them. But here, the VCS will require user intervention because it won't be able to figure out what to do.

It is better to avoid concurrent modifications on a file if you have to rely on two-way merge because it will be a slow manual process. Imagine 300 files requiring a simple merge...it would take ages to complete!

The Three-Way Merge

Let's forget for a second about version control, and go back to this "third person" looking into the two files we modified: How can he figure out what happened by himself?

He can look into the original version of the file we both used as starting point (Figure 2):

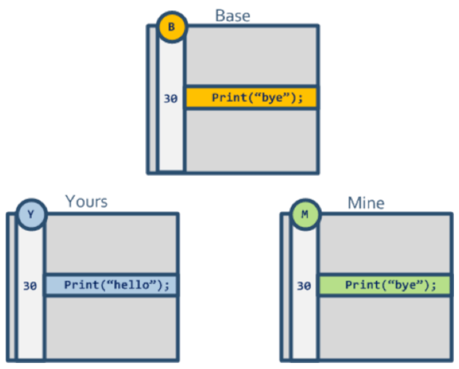


Figure 2.

Then, looking at how the file was originally ("base common ancestor" or simply "base"), he can figure out what to do.

Based on Figure 2, only one developer actually changed line 30, so the conflict can be manually resolved: Just keep "Yours" as the solution, so it will say: `Print("hello");`

This is how three-way merge helps: It turns a manual conflict into an automatic resolution. Part of the magic here relies on the VCS locating the original version of the file. This original version is better known as the "nearest common ancestor." The VCS then passes the common ancestor and the two contributors to the three-way merge tool that will use all three to calculate the result.

Using two-way merge only, the lines modified by two developers will require manual intervention, while everything else will be automatically merged. With three-way merge, it is possible to run a painless merge involving hundreds of files.

A Manual Merge Scenario

Let's now check a slightly more complex case, as shown in Figure 3:

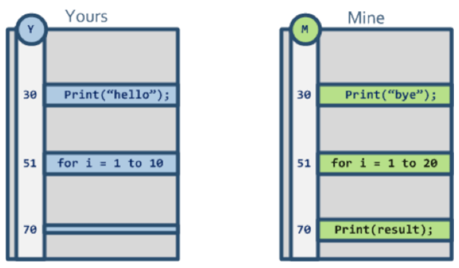


Figure 3.

Comparing the two files side by side, we can see that there are three lines with differences:

- Line 30: with the same conflict we had before.
- Line 51: with a `for` loop being modified.
- Line 70: where we don't know whether "yours" removed some code or "mine" added it.

Let's now look at the common ancestor to be able to properly solve the conflicts (Figure 4):

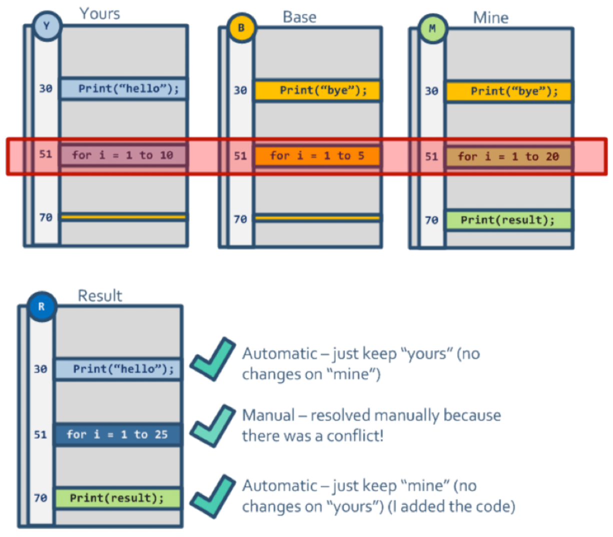


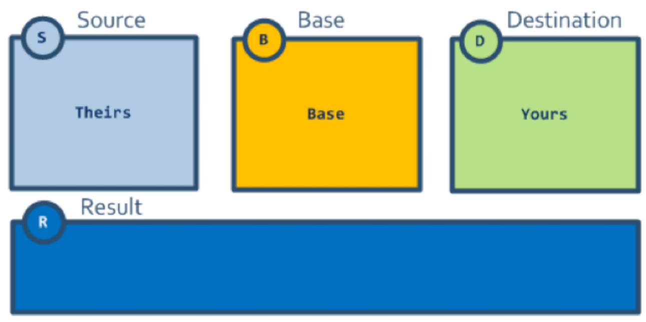
Figure 4.

- The conflict on line 30 can be automatically solved and the "yours" (source contributor) will be kept as result because only one contributor modified.
- The conflict on line 70 can also be automatically solved to "mine" (destination contributor) because it is clear now that the line has been added and it wasn't there before.
- The conflict on line 51 needs manual resolution: You need to decide whether you want to keep one of the contributors, the other, or even modify it manually.

And this is basically how three-way merge works.

Three-Way Merge Tool Layout

When you run a three-way merge tool, the typical layout of the tool is as illustrated in Figure 5:



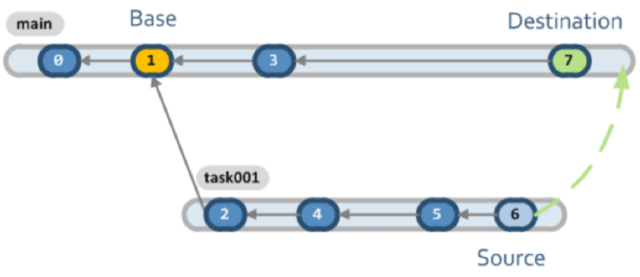


Figure 5.

Good three-way merge tools show four panels:

- "Theirs" (the source of the merge, see the branch diagram in Figure 6), base, and "Yours" (the destination of the merge) in the upper panel.
- The result of the merge in the lower panel.

To me this four-panel representation of the three-way merge is the most intuitive, but some tools present this alternative layout with only three panels (Figure 6):

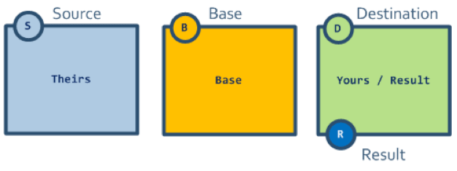


Figure 6.

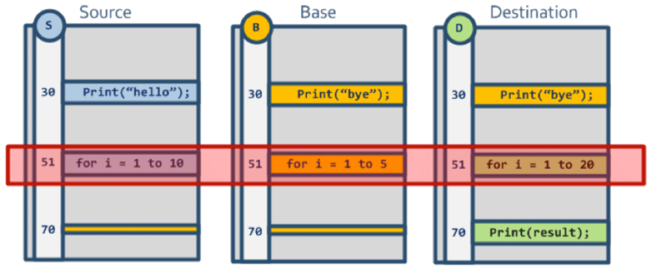
In this layout, the "destination/yours" (or working copy) and the result of the merge are displayed together.

The Importance of Merge Tracking

To run effective three-way merges, you not only need a good three-way merge tool, you need an effective merge engine in your version control tool.

In fact, part of the mission of version control should be to correctly calculate the common ancestor/base on any three-way merge. When people say "git is very good at merging," what they mean is "git is very good tracking the merge history, hence calculating the common ancestor for each file." In my VCS work, we put a lot of effort into the merge engine and calculating the nearest common ancestor.

Let's go back to the three-way merge with a manual conflict that we just solved, and let's check out the branching structure (well, at least one very simple branching structure); see Figure 7:



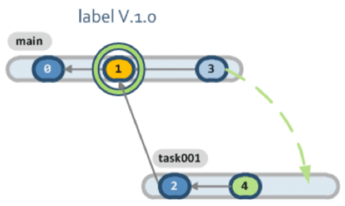


Figure 7.

- Changeset 3: someone working on the "main" branch performed the change of the `Print("hello")` line
- Changeset 4: meanwhile, on branch "task001," you were doing the addition of the `Print(result)` at line 70.
- And you both modified line 51.

Now, you want to merge the latest changes coming from "main" into your branch "task001." The version control system will find the nearest common ancestor of changesets "3" and "4" and it will use the graph above. The result in this simple case is changeset "1." The "base" version will be retrieved from changeset "1" to do the merge.

Once you solve the manual conflict on line 51, you will be checking in on the branch "task001" and creating a new changeset "5" as in Figure 8:

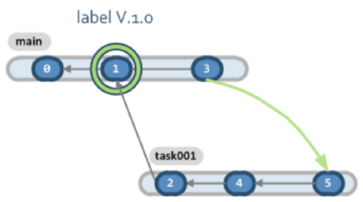


Figure 8.

Now development continues; somebody will be creating more changes on "main" while you perform a new checkin on "task001." And then you decide you have to merge "task001" back to "main" (Figure 9):

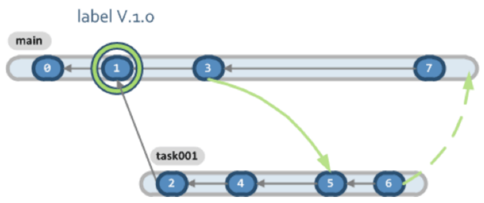


Figure 9.

The version control system will have to calculate the base/common ancestor between "6" and "7."

The common ancestor will be "3" as Figure 10 shows:

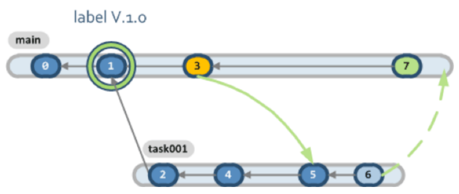


Figure 10.

Note that "3" is the common ancestor because the version control is considering the merge that happened between "3" and "5," which you completed before.

What is the benefit of this merge tracking?

Well, if the merge link between "3" and "5" wasn't tracked (as used to happen with old version control systems), then the base would be "1" again, and you would have to again solve the manual conflict you already solved before. However, if the version control system does its job correctly, the ancestor will be identified as "3" and you won't have to waste time on conflicts you already solved.

Now, what would happen here with two-way merge? You would have to solve *every* difference manually because in two-way merges, every single modification is a conflict since the merge facility doesn't have a way to solve conflicts automatically.

Conclusion

Often, the questions regarding three-way merge are asked by developers using version control systems lacking good merge tracking such as CVS, Microsoft Visual SourceSafe, and even old versions of Subversion.

Understanding how three-way merge works and why it is so important to have a good merge engine like those in new distributed version control systems is key when looking for a replacement to an aging SCM.

Pablo Santos is a blogger for Dr. Dobb's *and an expert on the operations of version control systems.*