www.plasticscm.com

advanced version control

POCKET GUIDE

All you need to know about branching, merging and DVCS

0 Introduction

Version control plays a key role in software development, and it is specially relevant for agile teams.

It is the cornerstone for best practices such us continuous integration, continuous delivery and devops.

Only using version control teams can implement the “collective code ownership” and enforce the concept of being “always ready to ship”.

There is one feature that makes all modern version control systems (Git, Mercurial, Plastic SCM) stand out from the previous generations: they excel on branching and merging.

The goal of this guide is to become a powerful tool for the expert developer by explaining the key concepts to master the most relevant merge techniques. Mastering branching and merging is the way to master version control.

Plastic SCM Team – Boecillo– October 2014

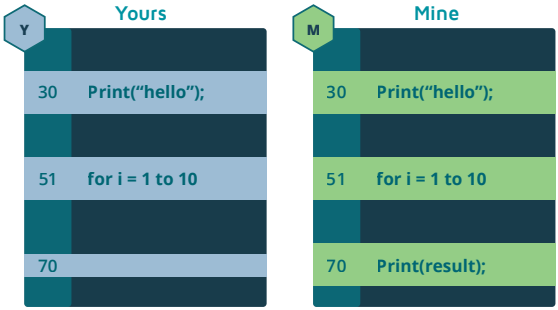
www.plasticscm.com/company/team

1 2-way merge

Many “arcane” version control systems were only capable of running 2-way merges (SVN, CVS). And that’s the reason why most developers fear merging.

All merges are **manual** in a 2-way merge and that’s why they’re slow, boring and error-prone.

Consider the following scenario:
“Did I add the line 70? Or did you delete it?”
There is no way to figure it out!
And this will happen for every single change if you merge using a 2-way merge tool. It is boring for a few files, painful for a hundred and simply not doable for thousands.

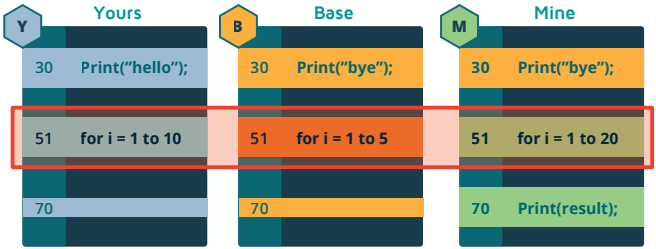


Find out more:
http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

2 3-way merge

All modern version controls (Git, Hg, Plastic SCM) feature improved merge tracking and enable 3-way merging.

3-way merge doesn't only compare "your copy to mine". It also uses the "base" (a.k.a. common ancestor) to find out "how the code was before our changes".
This changes everything! Now 99% of the merges will be automatic: no manual intervention required!



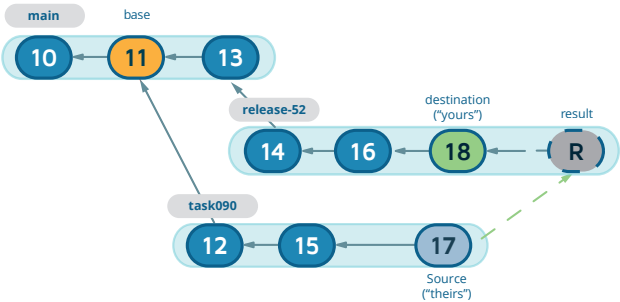
When you compare to the "base" conflicts are solved this way:

- *Line 30 - automatic – it will just keep yours.*
- *Line 70 - automatic – just keep mine (I added the line).*
- *Line 51 – manual – the user has to decide how to write the "for loop": loop from 1 to 10? 1 to 5? 1 to 20? Or maybe write something else?*

Find out more:
http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

3 merge contributors

When you merge between two branches you always deal with merge contributors:



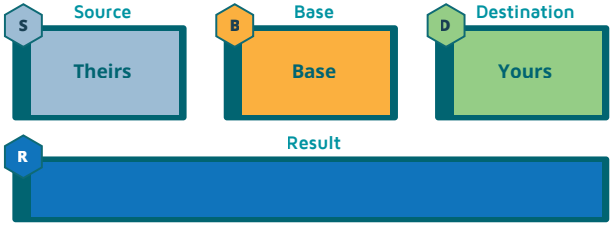
- *The developer needs to merge "17" and "18" and the result of the merge will be placed on branch "release-52".*
- *The version control calculates the "common ancestor" of "17" and "18". In our scenario the common ancestor (or base) is the changeset "11".*
- *The version control will launch the 3-way merge tool for each file in conflict. The conflicts will be found comparing "17" and "18" to "11".*

Once the merge is done the version control will create a "merge link" (the green arrow between "17" and "result") that will be used to calculate the common ancestor in upcoming merges.

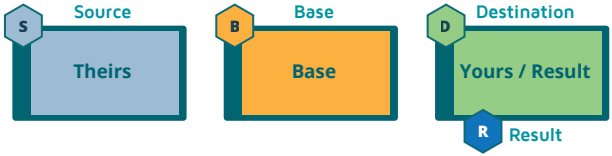
4 merge tool layout patterns

Almost all merge tools (Araxis, Xmerge, BeyondCompare, KDiff3) use one of the following patterns to handle the merge contributors:

They can use a “4 panel” layout as follows:



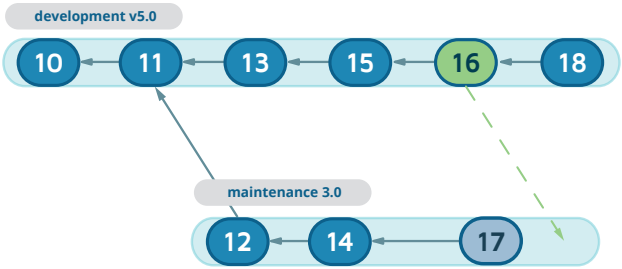
Or they can use a “3 panel” layout displaying “yours” and “result” together.



Once you understand this, merge tools won’t have secrets for you! :-)

5 cherry pick

How can we apply the fix of changeset “16” to the 3.0 branch?



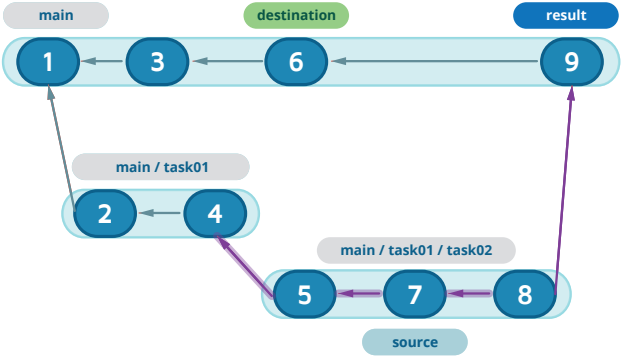
We can’t just merge “16” to “17” because then we’d apply all changes before “16” in branch 5.0 to 3.0. This would basically turn 3.0 into 5.0 which is definitely not what we want.

We just want to apply the “patch” of “16”, the changes made on “16” to the 3.0 branch.

This operation is known as “Cherry Pick”.

6 branch cherry pick

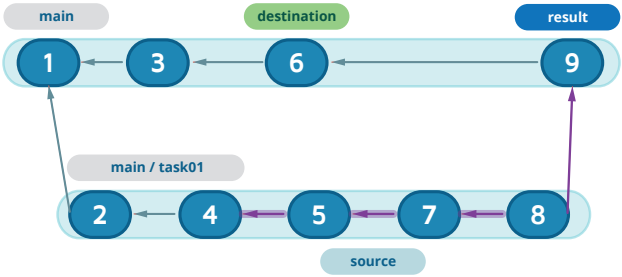
This is just a slightly modified “cherry pick” that allows you to apply a “branch level patch”: it will get the changes made on the branch but won’t take also the parent changes.



The merge in the figure takes the (4,8] interval: changesets “5”, “7” and “8” will be ‘cherry picked’ but not 2 and 4 as would happen with a regular merge.

7 interval merge

It is yet another way to run a cherry pick. This time the developer selects the beginning and end of the merge interval. This way he chooses exactly what needs to be picked to merge.



The scenario in the figure will get the changes inside interval (4, 8], which means only “5”, “7” y “8” will be taken.

8 subtractive merge

It is very powerful but you need to handle it with care.
It is very important to understand it is not just a “revert”.
You shouldn’t find yourself using subtractives on a regular basis: it is just a tool for special situations.

Look at the figure and consider we need to delete the change done by changeset “92” but keeping “93”, “94” and “95”.

We can’t just revert to “92” since we’d lose “93”, “94” and “95”.

What subtractive does is the following: 96=91–92+93+94+95.

It is an extremely powerful tool to “disintegrate” tasks, but you really need to know what you’re doing.

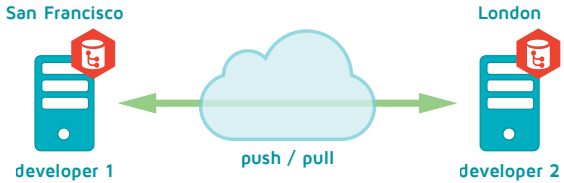
DVCS (distributed version control system) is the concept that took the industry by storm in the last decade.

Thanks to the new breed of tools: Git, Mercurial and our beloved Plastic SCM, version control is no longer considered a commodity and it is now seen as a competitive advantage.

With DVCS, teams don't depend anymore on a single central repository (and central server) since now there can be many clones and changes are "pushed and pulled" among them. In the case of Plastic SCM, there can be even partial clones.

Now teams working away from the head office don't have to suffer slow connections anymore, solving one of the classic issues in globally distributed development.

DVCS brings freedom and flexibility to design the repository and server structure.

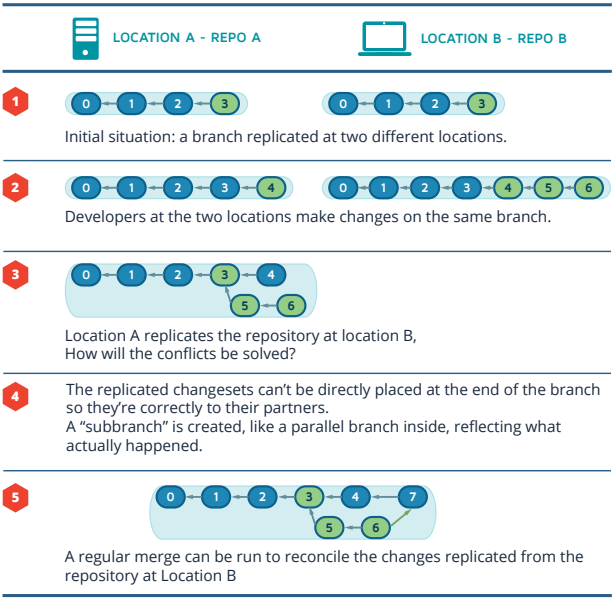


Learn more about the history of version control:
http://www.plasticscm.com/version-control-history.html

10 solving distributed conflicts

What happens when two developers work on the same branch on different repository clones? How will they reconcile the concurrent changes?

The figure below explains it step by step:



11 about us

Códice Software started in 2005 to develop Plastic SCM – a high performance distributed version control system for really advanced teams.

200 sprints and more than 600 releases later, Plastic SCM helps teams in more than 20 countries build better software. Teams in well-known companies like Microsoft, Samsung, Pantech, HP, Mapfre, DHL and TellTale. They all have something in common: they need the best branching and merging system, high performance, high scalability and distributed development.

Videogame studios around the world are currently switching to Plastic SCM because it is the only DVCS able to handle huge files, locks and combine centralized and distributed development.

Maybe the code of your favorite videogame is already controlled by Plastic SCM... :-)

Back in 2013 we released SemanticMerge, the world's first 3- way merge tool able to "understand your code".
www.semanticmerge.com

If you want to learn more about branching and merging and Plastic SCM, take a look at **"the merge machine"**.
www.plasticscm.com/mergemachine/index.html

plasticscm

Parque Tecnológico de Boecillo
Edificio Centro, 103
47151 Valladolid - SPAIN

sales@codicesoftware.com
support@codicesoftware.com