



Version Control, DevOps and Agile Development with Plastic SCM

Version Control, DevOps and Agile Development with Plastic SCM

The Plastic SCM book

Pablo Santos Luaces

Version 1.17, 2020-08-13

TABLE OF CONTENTS

Preface	1
About this guide	1
Conventions used in this book	2
Welcome	3
Starting up	5
Let's map a few concepts	5
A perfect workflow	5
Key steps	6
Workflow dynamics	8
How to implement	10
Where are you now?	11
Are task branches mandatory in Plastic SCM?	11
What is Plastic SCM	11
Why would someone consider Plastic?	12
Who is using Plastic?	13
What is version control	13
Plastic SCM Editions	14
How to install Plastic	14
If you are joining an existing project	14
If you are tasked to evaluate Plastic	15
Detailed installation instructions	15
How to get help	15
Command line vs. GUI	16
A Plastic SCM primer	17
Get started	17
3 key GUI elements	18
Workspace Explorer	18
Pending Changes	20
Branch Explorer	21
There are other views but...	23
Listing repos on different servers	23
Create a repo	25
Create a workspace	25
Adding files	27
Ignoring private files with ignore.conf	28
Initial checkin	29
Checkin changes	29
Undoing changes	30
Create a branch	30
Differing changesets and branches	34
Differing from the command line	34
Differing from the GUIs	35
Merge a branch	38

Merging from the GUI	39
Merging from the command line	40
Learn more	41
Annotate/blame a file	42
Annotate/blame from command line and GUI	43
You annotate a given revision of a file	44
Push and pull	44
Learn more	45
One task - one branch	47
Branch per task pattern	47
Branch naming convention	48
Task branches are short	50
Task branches are not feature branches	50
Keep task branches independent	51
What if you really need tasks to depend on each other	52
Techniques to keep branches independent	52
Checkin often and keep reviewers in mind	55
Antipattern 1: Checkin only once	55
Antipattern 2: Checkin for yourself	55
Checkin for the reviewer	56
Objection: But... you need to be very careful with checkins, that's a lot of work!	57
Task branches turn plans into solid artifacts that developers can touch	57
Handling objections	58
Task branches as units of change	60
What happens when a task can't be merged automatically?	60
We don't delete task branches	63
A finished task must be ready to be deployed	64
Feature toggles	65
Review each task	65
How we started reviewing every single task	66
Reviews are crucial to prevent code from rotting	66
Reviews to find bugs	67
How many reviewers?	67
How to actually do the reviews	67
Validation – exploratory tests on each task	67
A short intro to Exploratory Testing	68
Validation	68
A small story on our experience with Exploratory	69
Some extra pros of task branches	69
Colliding worlds: serial vs. parallel development	69
Code is always under control	71
Keep the main branch pristine	71
Have well-known starting points - do not shoot moving targets!	72
Enforce baseline creation	73
Stop bug spreading	73
Automated tests passing on each task branch	74

Automated tests are the gate to main	74
The Test Pyramid	74
Unit tests	75
Service/Integration tests	75
UI tests	75
Start small	76
Automated tests are a safety net	76
Every release is a release candidate	77
Extra testing – grouping releases	78
Be selfish with tests and clean code	78
Trunk-based development	79
What is trunk-based development	79
Task branches blend well with Trunk-based development	79
Why do I insist on using task branches instead of just doing checkin?	79
How do task branches blend with distributed development?	80
Automation, orchestration and mergebots	80
What about GitFlow?	81
How to learn more	81
Repo layout strategies	83
What is a repository?	83
Repository storage	83
Number of repos and maximum size	84
One project, one repo	84
Xlinks: Reusable components	85
Keep it simple - Don't overdesign your repo structure	86
Monorepos – don't divide and conquer	86
Submodules	87
Practical advice: Fantastic repos and when to create them	87
Keep it simple	87
Monorepo are fine	88
Sometimes you need Xlinks	90
Conclusion	90
Centralized & distributed	91
Centralized and distributed flavors and layouts	91
What is distributed and centralized	91
Plastic can do centralized and distributed	92
Is distributed better for branching and merging?	92
On-premises and Cloud	92
Mix distributed, centralized and Cloud	93
Multi-site	94
Recommended layout	94
Proxy / cache server	95
How replication works - push/pull	97
Globally unique changeset numbers	97
How push works	99
Why normal changeset numbers don't always match and we need GUIDs?	100

Push vs. pull	101
Handling concurrent changes on replicated repos	107
Concurrent changesets on different repos	107
Multi-head explained	107
Use pull to resolve concurrent changes on your repo	108
Sources of truth in distributed development	110
Single source of truth	110
Shared sources of truth	111
Our recommended option – single source of truth and not exact clones	114
Exact clones	116
Partial replicas	116
Replica in Plastic vs. Git	117
The dual lives of changesets	118
A trick to replicate just a single changeset from main	121
Merge history and partial replicas	122
Xlinks with distributed repos	124
Branching	127
Every repository starts with a main branch	127
Every changeset belongs to a branch	127
Creating branches is cheap	128
Branches have their own metadata	128
Branch hierarchies	129
Child branches	129
Top level branches	129
A meaningful branch hierarchy	130
Subbranches	131
Delete empty branches only	132
Changesets can be moved to a different branch	133
Diff branches	133
Merge and conflict resolution	135
Merge defined	135
Born to merge	135
Merge from a branch	135
Merge from a changeset	136
Merge contributors	137
Repositories are graphs of changesets	137
Arrow direction	138
Merge contributors: source, destination and base	140
Plastic always creates a changeset as result of the merge	143
Graphs with potential merge conflicts	143
2-way vs. 3-way merge	145
2-way merge: life before common ancestors	145
3-way merge	145
Layout of 3-way merge tools	148
Merge tracking	149
Calculating the common ancestor	150

Merging trees	152
Changeset-based merge tracking	156
Why merge tracking matters	158
Recursive merge - intro	161
Recursive merge - more than 2 ancestors	163
Recursive merge - why it is better than just choosing one ancestor	164
Plastic merge terminology	167
Directory merges	168
Directories are first class citizens	168
Diffing moves	171
Merging moved files	174
Change/delete conflict	176
Add/move conflict	177
Move/delete conflict	178
Evil twin conflict	180
Moved evil twin conflict	180
Divergent move conflict	182
Cycle move conflict	183
Conflict resolution order – directories before files	184
Automatic resolution of directory conflicts	185
Merge from and merge to	186
Removing changes – subtractive merge	187
What is a subtractive merge	187
When should you use subtractive merge?	187
How to undo a merge	187
Merging again a branch that was subtracted	189
Cherry picking	191
Cherry pick a changeset	191
Branch cherry pick	192
Interval merge	193
Conflicts during checkin – working on a single branch	194
Locking: avoiding merges for unmergeable files	198
Merging binary files	198
Lock to prevent concurrent changes	199
Locks only work in single branch	201
Plastic rebase vs. Git rebase	202
Workspaces	207
Full workspaces vs. partial workspaces	207
What is a workspace	207
Metadata of a workspace	209
Update and switch operations	210
Update	211
Switch	212
Update to a different repo	213
One repo many workspaces	213
Tuning the update operation in full workspaces	214

Cloaked	214
Readonly and writable	215
Tune EOLs	215
Finding changes	216
Looking for changes	216
Private files and ignore.conf	217
Detecting moves and renames	217
Hiding changes with hidden_changes.conf	219
Controlled changes - checkouts	219
Notifying added, deleted and moved	220
Notifying changes	222
When to use controlled changes	222
Advanced change tracking	224
Fast change detection with watchers	224
Advanced move detection	224
Switch branch with changes – why it is risky	224
What if you really want to override changes?	226
Full workspaces are always ready to merge	226
Partial workspaces	229
Configuring your partial workspace	229
Partial workspaces aren't in sync with a given changeset	230
Partial workspaces can checkin without downloading new changes	231
Fully checked and partially checked directories	232
How to convert a partial workspace into a full workspace and vice versa	233
Files that require merge during checkin	233
Xlinks in partial workspaces	235
How to learn more	237
More about Plastic SCM	237
Materials to learn more essential info	237
Git interop	238
References	238
DevOps with Plastic	238
DevOps driven by mergebots	238
DevOps driven by the CI system	239
Blog highlights	239
How we work	239
Great books to read	240
Appendix A: History of Plastic SCM	243
It all started with a dream... and a challenge	243
Getting real	243
A dream comes true	243
Códice and Plastic	244
Plastic 1.0, 2.0 and first international sales	244
VC time	245
Growing up – the road to 4.0	245
A mature solution	246

Appendix B: Pattern files	247
Filter pattern files	247
Rule types	247
Include / Exclude	250
Pattern evaluation hierarchy	250
Value matching pattern files	254
Rule Types	255
Pattern evaluation hierarchy	256
Appendix C: Alternative branching strategies and solutions to frequent problems	259
Branch per task with human integrator	259
Cycle definition	259
How it looks like in practice	261
Pros & cons	263
When to use it	264
Mainline only for unmergeable files	264
Release branches	264
Maintenance branches	265
A typical maintenance layout	266
Simple proposal for maintenance branches	267
When to merge maintenance branches back	268
Problems with maintenance branches	269
Where to apply bugfixes	270
How we do it with Plastic SCM	271
Conclusion	272
Branching for product variations	272
Typical challenges found in setups with multiple product branches	275
Proposed solution	275
No automated tests	277
Slow builds and slow tests	277
Big Bang integration	280
Appendix D: 2 principles for project management	283
Always have a fallback solution	283
Maximize visibility	284

PREFACE

About this guide

I don't know if you have ever had to go through the experience of writing a user guide, but I bet most of you did. It is a terribly challenging experience.

We struggled trying to figure out the best way to write our manuals for years. Sometimes you think no matter what you write, nobody is going to read it. From others you do get feedback and you are convinced even the smallest detail must be written down, because someone is going to find it or miss it.

Then there is the writing style. When we started back in 2005, we created something closer to a reference guide. It was painful to write, and I'm sure it was painful to read too. You know, we tried with general examples, with a professional and impersonal tone... that's what we thought was good at the time.

Later, we found out from our own blogging experience, that good examples and a really personal writing style worked better. It seems we humans are much better at extracting general cases from particular ones than vice versa. And, they are much easier to write too.

The thing is that, after almost one-and-a-half decades in the making, the original reference-style still persisted. So, we finally decided to go for a full rewrite of our user documentation, applying all the lessons we learned along the way.

I still love the old user manuals that came with computers in the 1980s. You know, they were not just about how to use the thing, but about really becoming a master. My first programming book was a user manual from my first Amstrad PCW 256. I miss these manuals. Now, you normally don't even get one when you receive a new laptop. I'm not sure if I'd really appreciate getting a copy of some C/C#/Javascript/whatever programming manual anyway, but it is all about nostalgia.

Anyway, this guide is not just about Plastic SCM. It is about how to become a better software developer mastering version control. It somehow tries to bring back some of the nostalgia from the old manuals, mixed with the "don't write a reference guide anymore" hard learned lesson.

It is prescriptive and opinionated sometimes, which now I think is a good thing, because we propose things, then it is your choice to apply them or not. If we simply leave everything as an exercise to the reader, if we stay independent, then there's not much value for you to extract as a reader. You can agree with us, or you can disagree, but both will be valuable takeaways.

For years we thought it would be too pretentious to tell teams how they should use our tool, how they should implement version control. We are a small team after all. Who are we to tell you how you should do your development? But, as I said, we learned that you choose our product not only because of the binaries you download, but to save hours of expert-seeking efforts and benefit from our own experience. We build a version control platform; we work with hundreds of teams around the world, we

might be mistaken, but you can definitely benefit from our mistakes and wise choices.

The following guide is our best attempt to put together all we know about version control and how to implement it, written in the most concise and personal way we know. We hope you enjoy it and find it helpful, because that's the only reason why we wrote it.

Conventions used in this book



This element indicates a note.



This element indicates a tip.



This element indicates an important text.



This element indicates a caution or warning.

WELCOME

Welcome to Plastic SCM!

If you are reading this, chances are you are interested in mastering version control and going beyond the most common tools and practices.

This guide will help you understand what Plastic SCM is, the underlying philosophy and best practices it is built upon, and will help you really choose if it is the right tool for you, or master it if you already decided to use it.

STARTING UP

Let's map a few concepts

Before we start, I'd like to compare a few concepts from Plastic SCM to other version control systems, so everything will be much easier for you to understand in the next sections.

Plastic SCM	Git	Perforce	Comment
Checkin	Commit	Submit	<i>Checkin</i> is what we call the action of sending changes to the repository.
Changeset	Commit	Changelist	Each time you checkin, you create a new changeset.
Update	Checkout	Sync	Download content to the local working copy. This happens, for instance, when you switch to a different branch.
Checkout	---	Edit	Simply tell Plastic that a file is going to be changed. It is not mandatory and you can skip this step. But, if you are using locks (you want exclusive access to a file), then checkouts are mandatory.
main	master	main	A.k.a. <i>trunk</i> in Subversion. The main branch always exists in a repository when you create it. In Plastic, you can safely rename it if needed.
Repository	Repository	Depot	The place where all versions, branches, merges, etc. are stored.
Workspace	Working tree	Workspace	The directory where your project is downloaded to a local disk and you use to work. In Git, the repo/working copy are tightly joined (the repo is inside a .git directory in the working copy). In Plastic, you can have as many workspaces as you want for the same repo.

Now you might be wondering: Why don't you version control designers simply use the same consistent naming convention for everything? The explanation could be really long here, but... yes, just to drive you crazy 😊

A perfect workflow

It is a good idea to start with a vision of what is achievable. And that's exactly what I'm going to do. I'm going to describe what we consider to be the state of the art in development workflows, what we consider to be *the perfect workflow*.



Key steps

Task

It all starts with a task in your issue tracker or project management system: your Jira, Bugzilla, Mantis, Ontime, or your own home-made solution. The key is that **everything you do in code** has an associated task. It doesn't matter whether it is a part of a new feature or a bugfix, create a task for it. If you are not practicing this already, start now. Believe me, entering a task is nothing once you and your team get used to it.

Task branch

Then you create a branch for the task. Yes, one branch for each task. No worries.

Develop

You work on your task branch and make as many checkins as you need. In fact, we strongly recommend to checkin very, very often, explaining each single step in the comments. This way you will be explaining each step to the reviewer, and it really pays off. Don't forget to add tests for any new feature or bugfix.

Code review

Once you mark your task as finished, it can be code reviewed by one of your colleagues. No excuses here, every single task goes to code review.

Validation

Then there is an optional step we do internally: validation. A colleague switches to the task branch and manually tests it. They don't look for bugs (automated tests take care of that) but rather, look into the **solution/fix/new-feature** from a customer perspective. The tester ensures the solution makes sense. This phase probably doesn't make sense for all teams, but we strongly recommend it when possible.

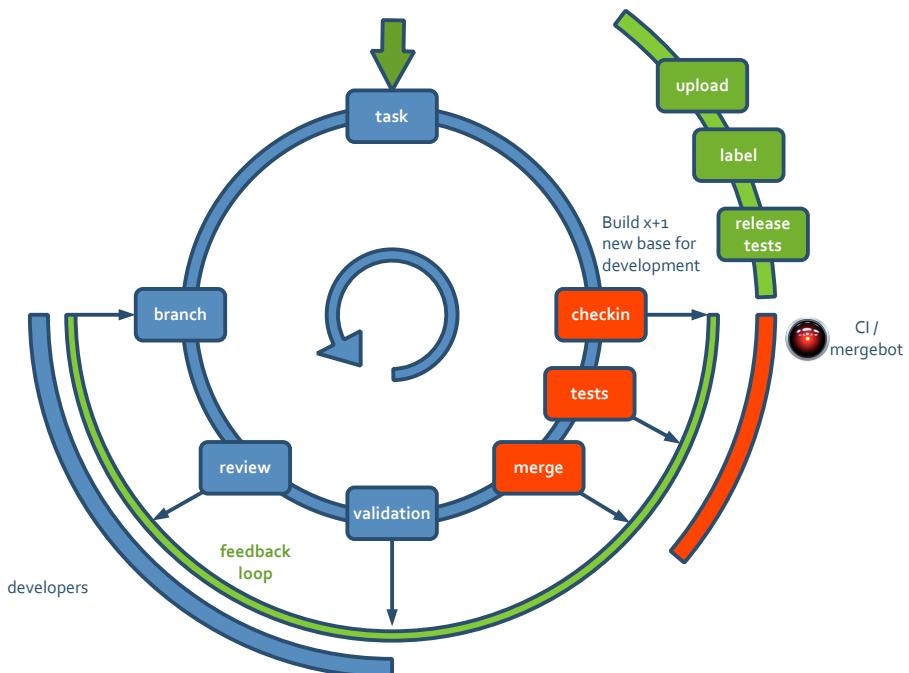
Automated testing and merge

Once the task is reviewed/validated it will be automatically tested. (After being merged but prior to the merge to be checkedin. More on that later). Only if the test suite passes the merge will it be confirmed. This means we avoid breaking the build at all costs.

Deploy

You can get a new release after every new task passes through this cycle or if you decide to group a few. We are in the DevOps age, with continuous deployment as the new normal, so deploying every single task to production makes a lot of sense.

I like to say the task goes through I cycle because I always image it as follows:



As you can see this *perfect workflow* aligns perfectly with the DevOps spirit. It is all about shortening the *task cycle times* (in Kanban style), about putting new stuff in production as soon as possible, about deploying new software as a strongly rooted part of your daily routine, and not an event as it still is for many teams.

Our secret sauce for all these is the cycle I described above and that can be stated as follows: ***The recipe to safely deliver to production continuously: short-lived task branches (16h or less) that are reviewed and tested before being merged to main.***

What is DevOps

DevOps is all about breaking silos and delivering quickly to production.

Leading organizations, such as the Amazons and Facebooks and the like of the world, deploy up to 100k times a day. (Of course, thanks to micro-services.)

It is all about superfast release cycles keeping the business in mind:

- Check out growth-hacking or any modern-day marketing practice: it encourages continuous experimentation, A/B testing, measuring and making data-driven decisions.
- To iterate fast, respond fast and adapt quickly, organizations need DevOps.

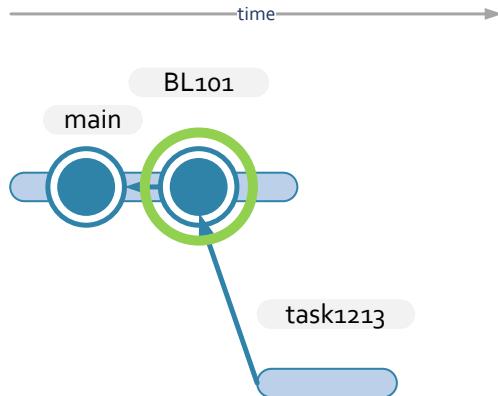
Test automation, continuous integration and delivery are just enablers to achieve the ultimate business goal.

Our favorite books about DevOps is "**The DevOPS Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations**" it is really worth reading. Super easy, short and straight to the point.

Workflow dynamics

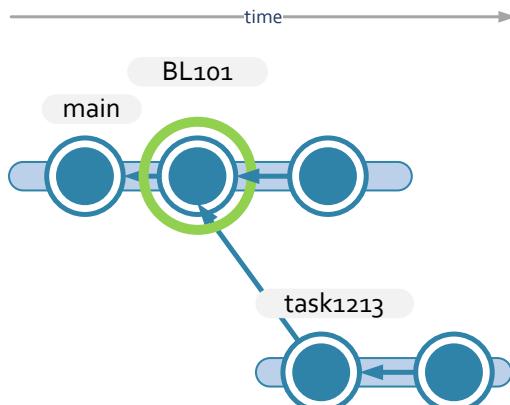
How does all the above translates to version control? The "One task - one branch" chapter is dedicated to this, but let's see a quick overview.

I'll start after you create a task for the new work to be done, and then you create a new branch for it:

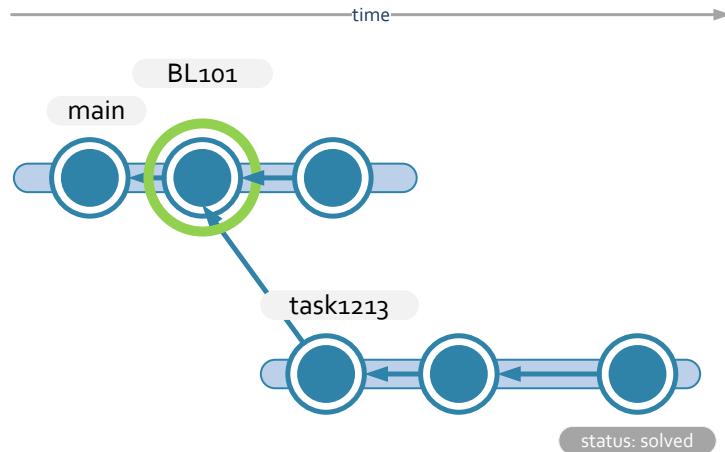


Notice that we recommend a very simple branch naming convention: a prefix (**task** in the example) followed by the task number in the issue tracker. You keep full traceability of changes and it is as simple as it can be. Remember, that Plastic branches can have a description too, so there is room to say what the task is about. I like to simply copy the task headline from the issue tracker and put it there, although we have integrations with most issue trackers that will simply do that automatically if you set it up.

Then, you start doing checkins of the changes you do to implement your task. Your task will look as follows (in fact, if it is a long task taking a few hours, you should have way more than a couple of checkins). Notice that, meanwhile, the main branch also evolved (I'm just doing this to make the scenario more realistic).



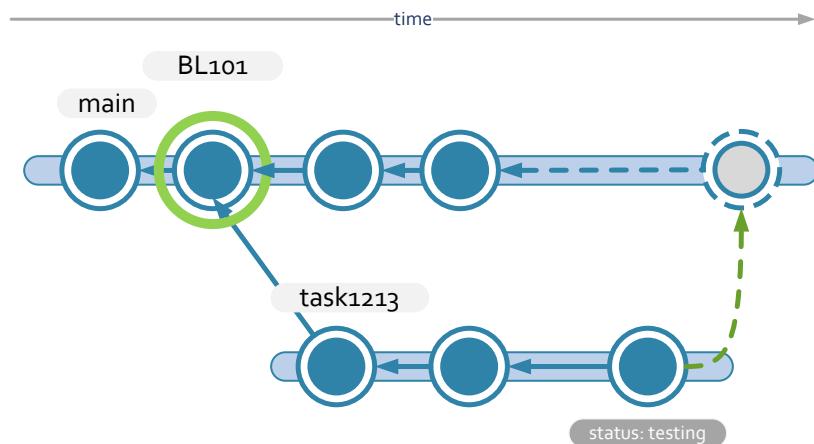
After more work, your task is completed in terms of development. You mark it as *solved* with an attribute set to the task. Alternatively, you can simply mark it as finished in your issue tracker. It all depends in the details of your particular tool set and how you will actually end up implementing the workflow.



Now it is the turn for the reviewer to look at your code and see if they can spot any bugs, or if there is something in your coding style or particular design that should be changed.

As I described earlier, once the review is ok, someone could do a validation too.

If everything goes well, the automatic part kicks in; you code merges and is submitted to the CI system to build and test.

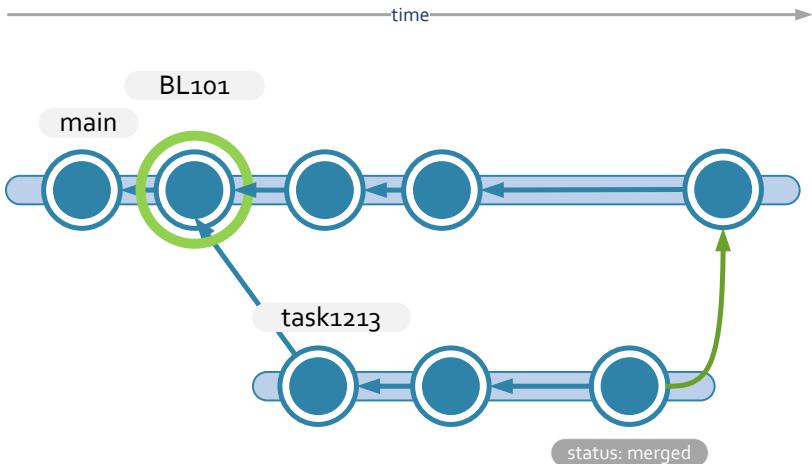


Two important things to note here:

- Note the dashed arrows saying that the new changeset is *temporary*. We prefer not to checkin the merge before the tests pass. This way we avoid broken builds.
- We strongly encourage you to build and pass tests on the merged code. Otherwise, tests could pass in isolation in `task1213` but ignore the new changes on `main`.

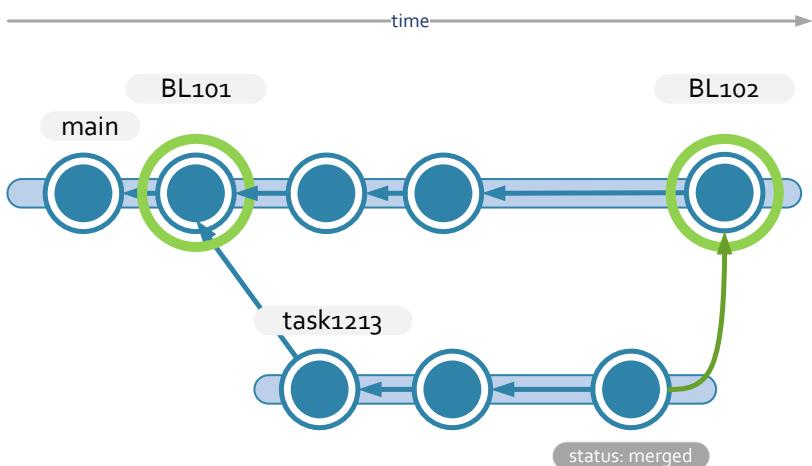
Then the CI will take its time to build and pass tests depending on how fast your tests are. The faster, the better, by the way, for obvious reasons.

If tests pass, the merge will be checkedin (confirmed) and your task branch will be now as follows:



Notice the attribute change too: it is now set to *merged*. Again, instead of an attribute set to the branch, you could track the status in your issue tracker (or on both, as we do internally).

Is the new release ready to be deployed? If it is, the new changeset on main is labeled as such and the software deployed to production (or published to our website as we do with Plastic SCM itself):



How to implement

I guess by now that you might have a bunch of questions:

1. *What parts of what you describe are actually included in Plastic?*

Easy, Plastic is a version control, so we help you with all the branching and merging, (even the visualization you saw in the example) are part of the product. Plastic is not a CI system but it is a *coordinator* (thanks to mergebots, more on that later), so it can trigger the builds, automate the merges, perform the labelling and more.

2. *What CI systems does Plastic integrate with?*

Virtually all of them because Plastic provides an API. But out of the box it integrates with Jenkins,

Bamboo and TeamCity.

3. You mentioned you have a **CI coordinator**. Can't I simply just go and plug my own CI?

Of course, you can plug in your own CI and try to implement the cycle above. We have done that dozens of times with customers.

4. Do I need a CI system and an issue tracker to implement task branches?

Well, if you *really* want to take full advantage of the cycle above, and full automation, you should. But you can always go manually: you can create task branches, work on them, and have someone in the team (integrator/build-master) doing the merges. That's fine, but not the state of the art. We [used to do it this way](#) [<https://www.plasticscm.com/download/help/trunk>] ourselves some time ago.

Where are you now?

You might be ahead of the game I described above. If that's the case, we still hope you can fit Plastic into your daily work.

If that's not the case, if you think this is a good workflow to strive for, then the rest of the book will help you figure out how to implement it with Plastic SCM.

Are task branches mandatory in Plastic SCM?

Of course, they are not. Plastic is extremely flexible, (That's why we named it what we did after all), and you are free to implement any pattern you want.

But, as I mentioned in the introduction, I'm going to be prescriptive and really *recommend* what we truly believe to work.

We strongly trust task branches because they blend very well with modern patterns [like trunk based development](#) [<https://www.plasticscm.com/download/help/trunkandtaskbranches>] but we'll cover other strategies in the book.

For instance, many game teams prefer to work on single branch, always doing checkin to main (especially artists), and that's fine.

What is Plastic SCM

Plastic SCM is a full version control stack designed for branching, merging, speed, visualization and ease of use.

Plastic is perfect to implement "task branches", our favorite pattern that combines really well with trunk-based development, DevOps, Agile, Kanban and many other great practices.

When we say "full version control stack" we mean it is not just a barebones core. Plastic is the repository management code (the server), the GUI tools for Linux, macOS and Windows, command line, diff tools, merge tools, web interface, and cloud server. In short, all the tools you need to manage the versions of your project.

Plastic is designed and evolved to handle thousands of branches, automate merges other systems can't, provide the best diff and merge tools, (we are the only ones providing semantic diff and merge), work centralized or distributed (Git like, or SVN/P4 like), to deal with huge repos (4TB is not an issue) and be as simple as possible.

It is a big task for a small team, and that's why we have been around for almost a decade and a half.

Plastic is not built on top of Git. It is a full stack. It is compatible with Git, but doesn't share the code base. It is not a layer on top of Git like GitHub, GitLab, BitBucket, the Microsoft stack and other alternatives. It is not just a GUI either like GitKraken, Tower and all the other Git GUI options out there. We build the entire stack. And, while this is definitely hard, it gives us the know-how and freedom to tune the entire system to achieve things others simply can't.

Finally, explaining Plastic SCM means introducing the team that builds it: we are probably the smallest team in the industry building a full version control stack. We are a compact team of developers with, on average, +8 years of experience building Plastic. It means a low turnover, highly experienced group of people. You can meet us [here](https://www.plasticscm.com) [<https://www.plasticscm.com>].

Plastic is not something we develop to sell you some other service. We don't want to capture users to sell you issue trackers, cloud storage, or build a social network. Plastic is the thing. It is our final goal. It is what makes us tick and breathe. It pays our bills and salaries. And we put our souls into it to make it better.

Why would someone consider Plastic?

You have other version control options; mainly the Git ecosystem and Perforce.

You can go for bare Git, choose one of the surrounding Git offerings from corporations like Microsoft and Atlassian, or select Perforce. You can alternatively stick to SVN, CVS, TFS, Mercurial, or ClearCase... but we normally only find users moving away from these systems, not into them anymore.

Why then would you choose Plastic instead of the empire of Git or the well-established Perforce (especially since they are in gaming industry)?

Plastic is not a "mass-produced" option (if the term can apply to the software industry). It is a special piece of software meant to be used by high-performance teams who want to get the best out of version control. Simply put, teams that need to do things differently than their competitors (which already went to one of the mainstream alternatives).

What does this mean exactly?

- Do you want to work distributed like Git, but also have teams of team members working centralized? Then, Git can't solve your issue because it can't do real centralized, and Perforce is not strong in distributed.
- Does your team need to handle huge files and projects? This is very common in the gaming industry. Then Git can't do it. Perforce is a good option but it lacks the super-strong branching and merging of Plastic (again, not good at doing distributed).
- Do you regularly face complex merges? Then, neither of these two can match Plastic. By design, we can handle merge cases out of the scope of Git and Perforce. Besides, we package (we develop them!) the best diff and merge tools you can find, including the ability to even merge code moved across files (semantic diff and merge, our unique technology).
- Do you need fast Cloud repos located as close to your location as possible? Plastic Cloud lets you choose a datacenter from a list of more than 12.
- Finely-grained security? Plastic is based on access control lists. Git can't do this by design. Perforce is good at this, but doesn't reach the access control list level.
- Performance: Plastic is super-fast, even with repos that break Git or slow down Perforce. Again, a design choice.

- Extraordinarily good support: Our support is part of the product. This is how we see it. Plastic is good technically, but what really makes it beat our competitors is how we care about users. We answer more than 80% of questions in less than one hour. 95% in under eight hours. And you can talk to real team members, not just a call center, but real product experts who will help you figure out how to work better, will help you solve issues, or will simply listen to your suggestions.
- Responsive release cycles: We release faster than anyone else (several releases a week) to respond to what our users need.

The list goes on and on, but you probably get the idea.

Who is using Plastic?

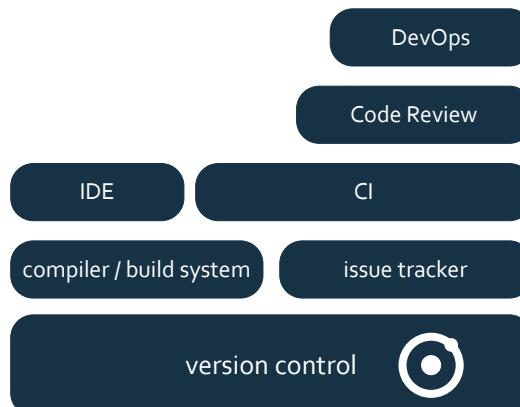
Yes, the question is: if Git is eating up the world of software development, who is selecting a niche player as Plastic?

The answer is Plastic is used by a wide range of teams: from companies with +3000 developers (three thousand) using Plastic, to small shops with just two coders starting up the next big game, and the entire range in between. Teams with 20-50 are very common.

What is version control

Version control is the operating system of software development. Yes, it might sound a little bit pretentious, but we spend our days developing version control so we better think we are up to a big challenge 😊

Jokes aside, we see version control as the cornerstone the rest of the software development sits upon. Version control is the one taking care of the project structure and contents that editors and IDEs will modify, compilers and the like will use to build, CI system to grab changes, trigger builds and deploy, issue trackers and project management tools to link to achieve full traceability, code review systems will access the changes to review and all of them will collaborate to deliver the software at a frequent pace (DevOps).



And version control is much more than a container. It is a collaboration tool. It is what team members will use to deliver each change. Properly used, it can shape the way the team works, enabling faster reviews, careful validation and frequent releases.

Version control is also a historian, the guardian of the library of your project. It contains the entire evolution of the project and can track incredibly useful information of why changes happened. Which is the key for improvement, you know: those who don't know their history, are condemned to repeat it.

We believe there is an extraordinary value in the project history, and we are only seeing the tip of the iceberg of what's to come in terms of extracting value from it.

Plastic SCM Editions

Of course, the different *editions* we offer in Plastic SCM are crystal clear in our minds ☺. But sometimes it seems they need some explanation. You can find all the detailed info about different editions [here](https://www.plasticscm.com/pricing) [<https://www.plasticscm.com/pricing>].

Here is a quick recap:

Type	Name	Who it is for
Paid and Free	Cloud Edition	<p>Teams who don't want to host their own Plastic SCM server.</p> <p>They can have local repos and push/pull to Cloud.</p> <p>Or they can directly checkin/update to Cloud.</p> <p>With the subscription, you get the storage, the managed cloud server, and all the Plastic SCM software.</p> <p>Cloud Edition is licensed per user in a dynamic way. It means if one month nobody uses Plastic, you don't pay for it, just for the repos stored in the cloud, if any.</p> <p>Cloud Edition is FREE up to a given amount of monthly users and GB of storage.</p>
Paid	Enterprise Edition	<p>Teams who host their own on-premises server.</p> <p>The subscription comes with all the software, server, tools plus a number of optimizations for high performance, integration with LDAP, proxy servers (the kind of things big teams and corporations need to operate).</p>

Plastic SCM is **licensed per user**, not server or machine. It means a developer can use a single license on macOS, Linux and Windows, and pays only one license. It also means you can install as many servers as you need without extra cost.

How to install Plastic

Installing Plastic SCM should be straightforward: just go to our [downloads page](https://www.plasticscm.com/download) [<https://www.plasticscm.com/download>] and pick the right installer for your operating system. Everything should be just *next, next, next* except if you are on Linux. If that's the case, just follow the instructions (2 commands, not that hard) for your specific distro flavor.

If you are joining an existing project

If you are just joining an existing project, you probably just need the client.

- Ensure you understand whether you are going to work centralized (direct checkin) or distributed

(local checkin to a local repo, then push/pull).

- **Check the Edition** your team is using and ensure you install the right one. If they are using Cloud Edition, you just need that. If they are in Enterprise, ensure you grab the right binary. It is not much drama to change later, but if you are reading this, you already have the background to choose correctly 😊
- If you are using Enterprise and you are going to work distributed (push/pull), you'll need to install a local server too. Just next, next, no worries. It is just the way we have to handle local repos.

If you are tasked to evaluate Plastic

Congrats! You are *the* champion of version control of your team now. Well, we'll help you move forward, but a few hints:

- **Check the Edition** that best suits your needs.
- You'll need to install a server. The default installation is fine for production, it comes with our **Jet storage** [<https://www.plasticscm.com/download/help/jetstory>] which is the fastest. If it is just you evaluating, you can install everything on your laptop, unless you want to run actual performance tests simulating real loads. But, functionality-wise, your laptop will be fine.
- You'll need a client too; both GUIs and CLIs come in the same package, not difficult.

Detailed installation instructions

Follow **this guide** [<https://www.plasticscm.com/download/help/plasticinstallation>] to get detailed instructions of how to install Plastic on different platforms.

How to get help

It is important to highlight that you can contact us if you have any questions. Just go to our **page support** [<https://www.plasticscm.com/support>] and select the best channel.

- It can be an email to support (you can do that during evaluation).
- Check our **forum** [<http://www.plasticscm.net>].
- Or even create a ticket in our support system if you prefer.
- You can also reach us on Twitter @plasticscm.

What is really important is that you know that we are super responsive. I mean, you will talk to the actual team (you can meet us on our **team page** [<https://www.plasticscm.com/company/team>]) and we care a lot about having an extraordinary level of response. Our stats say we answer 80% of the questions in under one hour, and more than 95% in under eight hours.

We can't stress enough how key this is for a product like Plastic SCM, that this will be a game changer for your team. It is not like we are a huge team hiding our experts behind several layers of first responders. We care a lot, we are small, and you'll have the chance to interact with the real experts. And it means not only simple questions, we are used to recommending to teams about the best way of working based on their particularities.

By the way, we are always open to suggestions too 😊

Command line vs. GUI

In many version control books there are many command line examples. Look at any book on Git and you'll find plenty of them.

Plastic is very visual because we spend lots of efforts not only in the core, but in the GUIs too. That's why traditionally most of our docs and blog posts were very GUI centric.

For this guide we're going to try to use the command line whenever it helps to better explain the underlying concepts, and we'll keep very *interface agnostic* and *concept focused* whenever possible. But, since GUIs are key in Plastic, expect some screenshots too ☺

A PLASTIC SCM PRIMER

Let's get more familiar with Plastic SCM before we cover the recommended key practices about navigating branching, merging, pushing and pulling.

The goal of this chapter is to get you started with Plastic so you learn how to run the most common operations that will help you during the rest of the book.

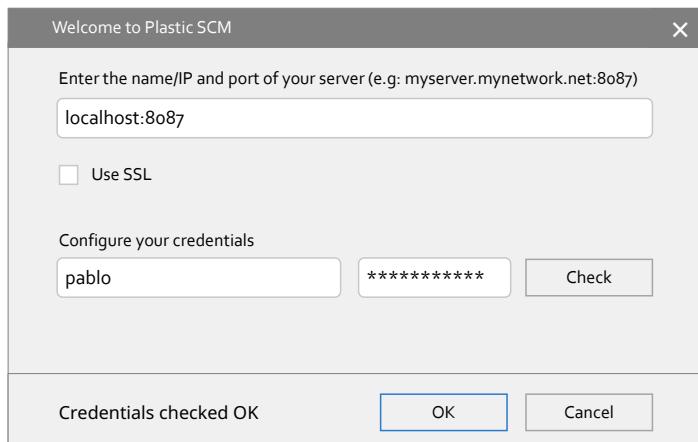
Get started

I'm going to assume you already installed Plastic SCM on your computer. If you need help installing, I recommend that you follow our "[version control installation guide](#)" [<https://www.plasticscm.com/download/help/plasticinstallation>].

The first thing you need to do is to select a default server and enter the right credentials to work on it.

If you use the command line, you can use `clconfigureclient` [<https://www.plasticscm.com/download/help/clconfigureclient>] for that.

If you use the GUI, the first thing you will see is a screen as follows:



Where you have to enter the server, decide if you want to connect through SSL, and then enter your credentials.

After that, the GUI guides you to create your first workspace connected to one of the repos on the server you selected.



About the GUI

As you saw in the previous figure, we'll be using a mockup to represent the GUI elements instead of screenshots of the actual Linux, macOS or Windows GUIs. I preferred to do this for two reasons: first to abstract this guide from actual GUI changes, and second to avoid certain users feeling they are excluded. I mean, if I take screenshots in Windows, then Linux and macOS users will feel like second class citizens. If I add macOS screenshots, Windows and Linux users will be uncomfortable. So, I finally decided just to show mockups that will help you find your way on your platform of choice.

You noticed that I entered `localhost:8087` as the server. Let me explain a little bit about this:

- For Cloud Edition users, their server will typically be just "local", without a port number. This is because every Cloud Edition comes with a built-in local server and a simplified way to access it.
- If you are evaluating Enterprise Edition and you installed a server locally in your machine, then `localhost:8087` will be the default. We chose port `8087` as the default one long ago.
- But if you are connected to a remote server in your organization, maybe your server will be something like `skull.codicefactory.com:9095`, which is the one we use.
- If you connect directly to a Plastic Cloud server, then it will be something like `robotmaker@cloud`, where `robotmaker` will be the name of your cloud organization instead.

Authentication modes.

Plastic supports a number of authentication modes, primarily LDAP, Active directory and user/password. Then there are other modes (Name) used for evaluation purposes. Learn more about auth modes [here](#) [<https://www.plasticscm.com/download/help/authmodes>].

3 key GUI elements

Before we move forward, I'd like to highlight the three most important elements you will use in the GUIs.

- Workspace Explorer (previously known as "items view").
- Pending Changes, a.k.a. Checkin Changes.
- Branch Explorer.

Workspace Explorer

This view is like a Windows Explorer or macOS Finder with version control information columns.

Plastic SCM						
Workspace Explorer	Path	Status	Size	Type	Author	Date
	- /home/pablo/wkspaces/wktest	Controlled		Directory	pablo	2018/12/28
	- art	Controlled		Directory	ruben	2018/11/29
	+ car	Private		Directory	pablo	2018/12/24
	- code	Controlled	Size	Directory	manu	2018/12/10
	- botlib	Controlled	Size	Directory	vio	2018/12/11
	foo.c	Controlled	21 KB	Text	miryam	2018/12/13
	bar.c	Controlled	12 KB	Text	pablo	2018/12/31

You can see the files you have loaded into your workspace and the date when they were written to disk, and also the changeset and branch where they were created (although I'm not including these last two in the figure above).

It is the equivalent of running a `cm ls` command.

```
cm ls
0 12/28/2018 14:49 dir br:/main .
0 11/29/2018 14:49 dir br:/main art
0 12/10/2018 18:23 dir br:/main code
```

The most interesting things about the Workspace Explorer are:

- It comes with a finder that lets you quickly locate files by name patterns. CTRL-F on Windows or Command-F on macOS.
- There is a context menu on every file and directory to trigger certain actions:

Plastic SCM						
Workspace Explorer	Path	Status	Size	Type	Author	Date
	- /home/pablo/wksp...	Add to source control		Directory	pablo	2018/12/28
	- art	Open		Directory	ruben	2018/11/29
	+ car	Diff		Directory	pablo	2018/12/24
	- code	Checkout		Directory	manu	2018/12/10
	- botlib	Checkin		Directory	vio	2018/12/11
	foo.c	Undo checkout		File	miryam	2018/12/13
	bar.c	Add to cloaked		File	pablo	2018/12/31
		Add to hidden changes				
		Rename				
		Cut				
		Paste				
		Annotate				
		View history				

Not all actions are enabled all the time since they are dependent on the context (for example, you can't check the history of a private file) but it gives you an idea of what you can do.

It is very useful to find a file and check its history or an annotate/blame and see where every line comes from.

Pending Changes

This "view" is so important that you can perform most of your daily operations from here. In fact, I strongly recommend to use this view and not the Workspace explorer to add files, and checkin all changes.

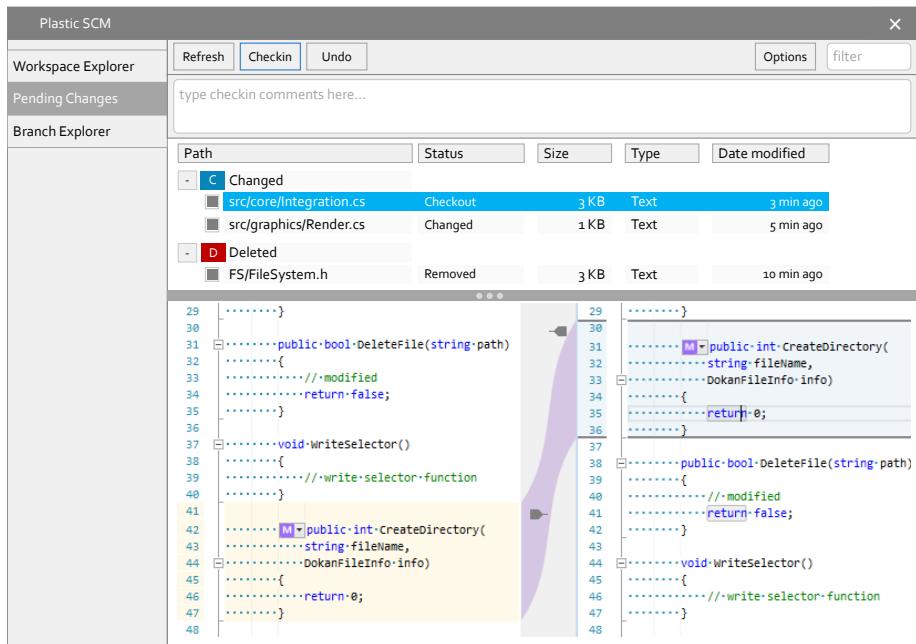
The layout of Pending Changes in all our GUIs is as follows:

The screenshot shows the Plastic SCM Pending Changes interface. At the top, there are buttons for Refresh, Checkin (highlighted in blue), Undo, Options, and filter. Below this is a section for checkin comments. The main area displays a table of changes:

Path	Status	Size	Type	Date modified
- [C] Changed				
src/core/Integration.cs	Checkout	3 KB	Text	3 min ago
src/graphics/Render.cs	Changed	1 KB	Text	5 min ago
- [D] Deleted				
FS/FileSystem.h	Removed	3 KB	Text	10 min ago
src/graphics/Render.cs	Removed Locally	1 KB	Text	5 min ago
- [M] Moved				
docu to documentation	Moved		Directory	23 min ago
inc to include	Moved Locally		Directory	15 min ago
- [A] Added and private				
src/graphics/lIcon.cs	Added	2 KB	Text	7 min ago
src/graphics/Bmp.cs	Private	1 KB	Text	1 min ago

- There are important buttons at the top to refresh and find changes (although you can also configure all GUIs to do auto-refresh), checkin and undo changes.
- Then, there is a section to enter checkin comments. This is the single most important thing to care about when you are in this view: ensure the comment is relevant. More about checkin comments later in the chapter about task branches.
- And finally, the list of changes. As you can see, Pending Changes split the changes in "changed", "deleted", "moved" and "added". Much more about all these options in the "[Finding changes](#)" section in the "Workspaces" chapter.

The other key thing about Pending Changes is the ability to display the diffs of a selected file:

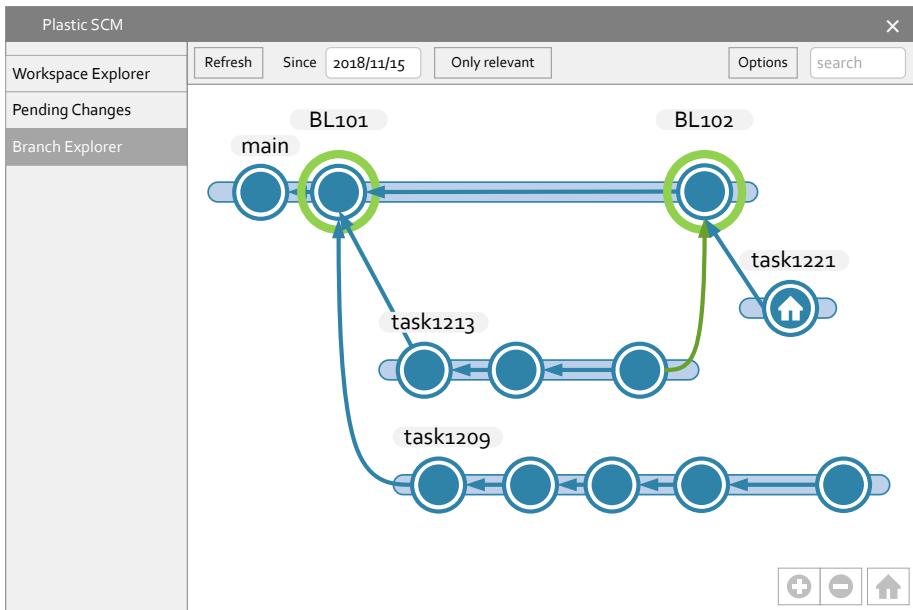


Plastic comes with its own built-in diff tool. And as you can see, it can do amazing things like even tracking methods that were moved. We are very proud of our SemanticMerge technology.

Most of the Plastic SCM actions you will do while developing are done from Pending Changes. You simply switch to Plastic, see the changes, checkin, and get back to work.

Branch Explorer

The Branch Explorer diagram is our crown jewel. We are super proud of it and the way it can render the evolution of the repos. The Branch Explorer (a.k.a. BrEx) view has the following layout:



Every circle in the graphic is a changeset. The changesets are inside containers that represent the branches. The diagram evolves from left to right, where changesets on the right are newer than the ones on the left. The actual Branch Explorer in the GUIs draw columns with dates to clarify this.

The green arrows are merges and the blue ones connect changesets with a "is parent of" relationship.

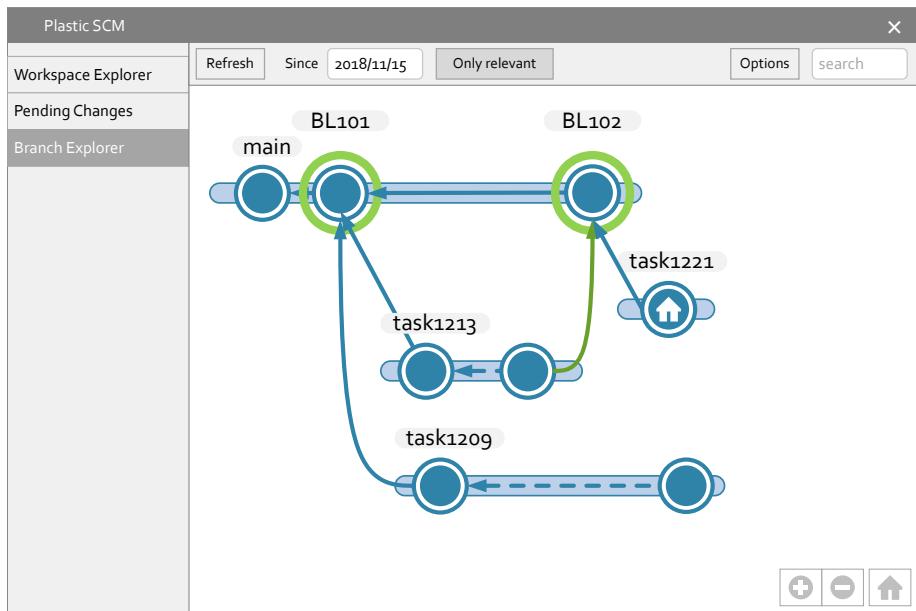
The green donuts surrounding changesets are labels (the equivalent to Git tags).

The Branch Explorer lets you filter the date from where you want to render the repo, and it has zoom, a way to search, and also an action button to show "only relevant" changesets. It also shows a "house icon" in the changeset your workspace is pointing to at any moment (this is true for regular Plastic, not Gluon as you will see in the "Partial Workspaces" section).

By the way, if you click "Only relevant", the Branch Explorer will compact as the following figure shows, since the "relevant changesets" are:

- Beginning of a branch.
- End of a branch.
- Labeled changeset.
- Source of a merge.
- Destination of a merge.

This way, it is very easy to make large diagrams more compact by hiding "non-relevant" changesets.



From the Branch Explorer context menus, you can perform a number of key actions. In fact, you can select every branch, changeset, label and right click on it to discover their actions.

Typical actions you can perform are:

- Switch your workspace to a different branch, changeset or label.
- Diff branches, changesets and labels.
- Push and pull branches.
- Merge branches and changesets.
- Actions to set attributes to branches.

There are other views but...

In my opinion, you can perform all the daily actions just using Pending Changes and the Branch Explorer, and maybe a little bit of the Workspace Explorer. If I had to remove everything except what was absolutely essential, Plastic would run just with these three.

There are other views like the Branches View, that lists branches in a more traditional way (a list), and Changesets View, and Attributes, Labels... but you can use the Branch Explorer instead in a more visual way.

Listing repos on different servers

Go to the command line and run:

```
cm repository list
```

In my case, the output of the command is this:

```
cm repository list
quake@192.168.221.1:6060
quake_replicated@192.168.221.1:6060
quake_from_git@192.168.221.1:6060
ConsoleToast@192.168.221.1:6060
PlasticNotifier@192.168.221.1:6060
asyncexample@192.168.221.1:6060
geolocatedcheckin@192.168.221.1:6060
p2pcopy@192.168.221.1:6060
udt@192.168.221.1:6060
wifidirect@192.168.221.1:6060
PeerFinder@192.168.221.1:6060
```

The `repository [list | ls]` command shows the list of the repos on your default server (remember, the one you connected to initially).

But, it also works if you list the reps of another server:

```
>cm repository list skull:9095
Enter credentials to connect to server [skull:9095]
User: pablo
Password: *****
codice@skull:9095
pnunit@skull:9095
nervathirdparty@skull:9095
marketing@skull:9095
tts@skull:9095
```

I used the command to connect to `skull:9095` (our main internal server at the time of this writing, a Linux machine) and Plastic asked me for credentials. The output I pasted is greatly simplified since we have a list of over 40 repos on that server.

You can also use `repository list` to list your cloud repos. In my case:

```
>cm repository list codice@cloud
codice@codice@cloud
nervathirdparty@codice@cloud
pnunit@codice@cloud
marketing@codice@cloud
plasticscm.com@codice@cloud
devops@codice@cloud
installers@codice@cloud
```

This is again a simplified list. This time the command didn't ask me for credentials since I have a profile for the `codice@cloud` server.

Well, you get the point; every server has repos, and you can list them if you have the permissions to do so.

For you most likely, a `cm repository list` will show something like:

```
cm repository list  
default@localhost:8087
```

By default, every Plastic installation creates a repo named `default` so you can start doing tests with it instead of starting from a repo-less situation.

From the GUIs, there are "repository views" where you can list the available repos. Something as follows:

The screenshot shows the Plastic SCM interface with the title 'Plastic SCM'. At the top, there are buttons for 'Refresh', 'Server' (set to 'localhost:8087'), 'New repo ...', and 'filter'. Below this is a table with two columns: 'Name' and 'Server'. The table contains three rows: 'default' (Server: localhost:8087), 'quake' (Server: localhost:8087), and 'doom' (Server: localhost:8087). The 'Server' column is currently selected.

Name	Server
default	localhost:8087
quake	localhost:8087
doom	localhost:8087

There is always a way to enter a different server that you want to explore, and also a button to create new repos if needed.

Create a repo

Creating a new repository is very simple:

```
cm repository create myrepo@localhost:8087
```

Where `myrepo@localhost:8087` is a "repo spec" as we call it.

Using the GUI is straightforward so I won't show a screenshot, it is just a dialog where you can type a name for the repo 😊

Of course, you can only create repositories if you have the `mkrep` permission granted on the server.

Create a workspace

A workspace is a directory on disk that you will use to connect to a repository. Let's start with a clean workspace.

If you run:

```
cm workspace create demowk c:\users\pablo\wkspaces\demowk
```

Plastic will create a new workspace named `demowk` inside the path you specified.

It is very typical to do this instead:

```
cd wkspaces  
mkdir demowk  
cd demowk  
cm workspace create demowk .
```

Interestingly, I didn't specify which repository to use for the workspace. By default, the first repo in the default server is used, so in your case it will most likely be your default repo. There are two alternatives to configure the repo:

```
cm workspace create demowk . --repository=quake@localhost:9097
```

Or, simply create it in the default location and then switch to a different repo:

```
cm workspace create demowk .  
cm switch main@quake
```

Workspaces have a name and a path for easier identification. And, you can list your existing workspaces with `cm workspace list`.

```
>cm workspace list  
four@MODOK          c:\Users\pablo\wkspaces\dev\four  
five@MODOK          c:\Users\pablo\wkspaces\dev\five  
tts@MODOK           c:\Users\pablo\wkspaces\dev\tts  
plasticscm-com@MODOK c:\Users\pablo\wkspaces\dev\plasticscm-com  
doc-cloud@MODOK     c:\Users\pablo\wkspaces\dev\doc-cloud  
plasticdocu_cloud@MODOK c:\Users\pablo\wkspaces\dev\plasticdocu_cloud  
mkt_cloud@MODOK     c:\Users\pablo\wkspaces\mkt-sales\mkt_cloud  
udtholepunch@MODOK  c:\Users\pablo\wkspaces\experiments\udtholepunch
```

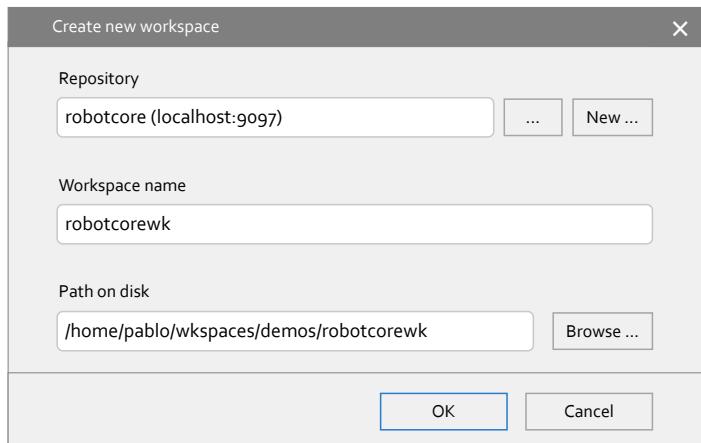
The previous is a subset of my local workspaces at my `modok` laptop 😊

From the GUI, there's always a view to list and create workspaces:

Plastic SCM - workspaces		
		X
Refresh	Create new workspace ...	filter
Name	Path	Repository
default	/home/pablo/wkspaces/default	default@localhost:8087
quake	/home/pablo/wkspaces/quake	quake@games@cloud
doom	/home/pablo/wkspaces/doom	doom@skull:9095

As you see, there can be many workspaces on your machine, and they can be pointing to different repos both locally and on distant servers. Plastic is very flexible in that sense as you will see in "[Centralized & Distributed](#)".

Creating a workspace from the GUI is straightforward, as the following figure shows:



Suppose you create a workspace to work with an existing repo. Just after creating the workspace, your Branch Explorer places the home icon in changeset zero. Initially, workspaces are empty until you run a `cm update` (or GUI equivalent) to download the contents. That's why initially they point to changeset zero.

Adding files

Let's assume you just created your workspace `demowk` pointing to the empty `default` repo on your server. Now, you can add a few files to the workspace:

```
echo foo > foo.c  
echo bar > bar.c
```

And then, check the status of your workspace with `cm status` as follows:

```
>cm status  
/main@default@localhost:8087 (cs:0 - head)  
  
Added  
Status      Size      Last Modified    Path  
Private     3 bytes   1 minutes ago   bar.c  
Private     3 bytes   1 minutes ago   foo.c
```

The GUI equivalent will be:

The screenshot shows the Plastic SCM GUI interface. In the top navigation bar, 'Plastic SCM' is selected. Below it, the 'Pending Changes' tab is active. The main area displays a table of pending changes:

Path	Status	Size	Type	Date modified
- A foo.c	Added and private	3 bytes	Text	1 min ago
- A bar.c	Added and private	3 bytes	Text	1 min ago

You can easily checkin the files from the GUI by selecting them and clicking "checkin".

From the command line, you can also do this:

```
cm add foo.c
```

Which will display the following status:

```
>cm status  
/main@default@localhost:8087 (cs:0 - head)
```

Added			
Status	Size	Last Modified	Path
Private	3 bytes	3 minutes ago	bar.c
Added	3 bytes	3 minutes ago	foo.c

Note how `foo.c` is now marked as "added" instead of simply "private" because you already told Plastic to track it, although it wasn't yet checked in.

If you run `cm status` with the `--added` flag:

```
>cm status --added  
/main@default@localhost:8087 (cs:0 - head)
```

Added			
Status	Size	Last Modified	Path
Added	3 bytes	5 minutes ago	foo.c

`foo.c` is still there while `bar.c` is not displayed.

Ignoring private files with ignore.conf

There will be many private files in the workspace that you won't checkin (temporary files, intermediate build files (the .obj), etc). Instead of seeing them repeatedly in the GUI or `cm status`, you can hide them by adding them to `ignore.conf`. Learn more about `ignore.conf` in "["Private files and ignore.conf"](#)".

Initial checkin

If you run `cm ci` from the command line now, only `foo.c` will be checked in, unless you use the `--private` modifier or you run `cm add` for `bar.c` first. Let's suppose we simply checkin `foo.c`:

```
>cm ci -c "Added foo.c"
The selected items are about to be checked in. Please wait ...
| Checkin finished 33 bytes/33 bytes [#####
Modified c:\Users\pablo\wkspaces\demowk
Added c:\Users\pablo\wkspaces\demowk\foo.c
Created changeset cs:1@br:/main@default@localhost:8087 (mount: '/')
```

Notice how I added a comment for the checkin with the `-c` modifier. This is the comment that you will be able to see in the Branch Explorer (and changeset list) associated to the changeset 1 you just created.

Checkin changes

Let's now make some changes to our recently added `foo.c`. Then run `cm status`:

```
>cm status
/main@default@localhost:8087 (cs:1 - head)

Changed
Status      Size       Last Modified    Path
Changed     1024 bytes   1 minutes ago   foo.c

Added
Status      Size       Last Modified    Path
Private    1024 bytes   5 minutes ago   bar.c
```

As you can see, `foo.c` is detected as changed, and the old `bar.c` is still detected as ready to be added (private).

From the GUI, the situation is as follows:

The screenshot shows the Plastic SCM interface with the 'Pending Changes' tab selected. The main pane displays a list of pending changes:

Path	Status	Size	Type	Date modified
- C Changed foo.c	Changed	1 KB	Text	1 min ago
- A Added and private bar.c	Private	1 KB	Text	5 min ago

And, you'll be able to easily select files, diff them and checkin.

Undoing changes

Suppose you want to undo the change done to `foo.c`.

From the GUI, it will be as simple as selecting `foo.c` and clicking "Undo".

Important: Only the checked elements are undone.

From the command line:

```
>cm undo foo.c  
c:\Users\pablo\wkspaces\demowk\foo.c unchecked out correctly  
  
>cm status  
/main@default@localhost:8087 (cs:1 - head)  
  
Added  
Status      Size      Last Modified      Path  
Private    1024 bytes   6 minutes ago    bar.c
```

Create a branch

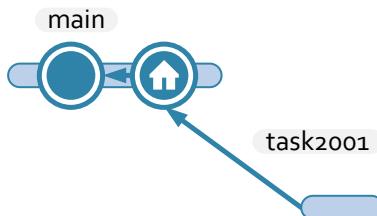
Now, you already know how to add files to the repo and checkin, so the next thing is to create a branch:

```
cm branch main/task2001
```

Creates a new branch child of `main`, and at this point it will be a child of changeset `1`.

There is a `--changeset` modifier to specify the starting point of your new branch.

Now your Branch Explorer will be as follows:

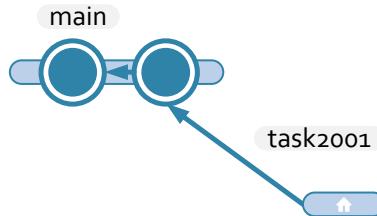


Your workspace is still on `main` because you didn't switch to the new branch.

Let's switch to the new branch this way:

```
> cm switch main/task2001
Performing switch operation...
Searching for changed items in the workspace...
Setting the new selector...
Plastic is updating your workspace. Wait a moment, please...
The workspace c:\Users\pablo\wkspaces\demowk is up-to-date (cset:1@default@localhost:8087)
```

And now the Branch Explorer diagram is:



With the home icon located in the new `main/task2001` branch that is still empty.

Let's modify `foo.c` in the branch by making a change:

```
>echo changechangechange > foo.c

>cm status
/main/task2001@default@localhost:8087 (cs:1 - head)

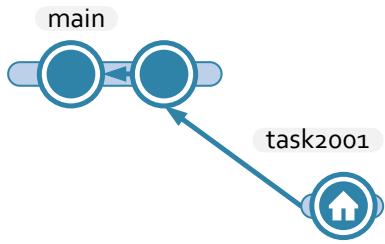
Changed
Status      Size       Last Modified     Path
Changed    21 bytes   8 seconds ago   foo.c

Added
Status      Size       Last Modified     Path
Private   1024 bytes  6 minutes ago   bar.c
```

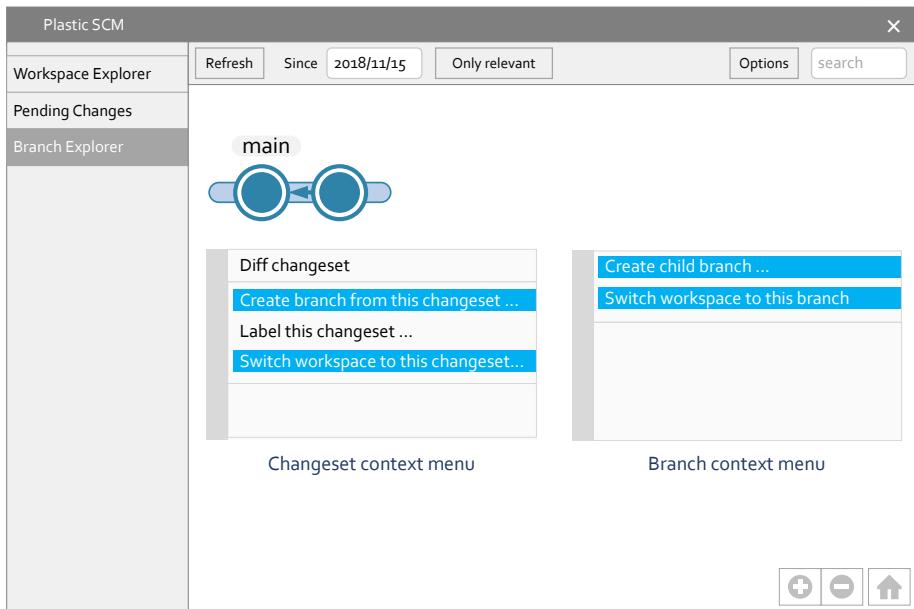
And then a checkin:

```
>cm ci foo.c -c "change to foo.c" --all
The selected items are about to be checked in. Please wait ...
| Checkin finished 21 bytes/21 bytes [########################################] 100 %
Modified c:\Users\pablo\wkspaces\demowk\foo.c
Created changeset cs:2@br:/main/task2001@default@localhost:8087 (mount: '/')
```

Now, the Branch Explorer will reflect the new changeset:

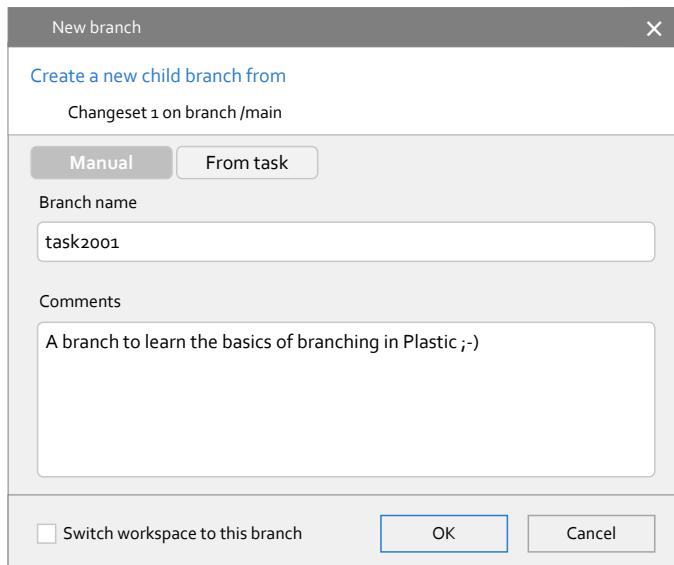


From the GUI, it is very simple to create new branches and switch to them:



The Branch Explorer provides context menus to create new branches from changesets and branches and switch to them. (I'm highlighting the interesting options only, although the context menus have many other interesting options to merge, diff, etc.).

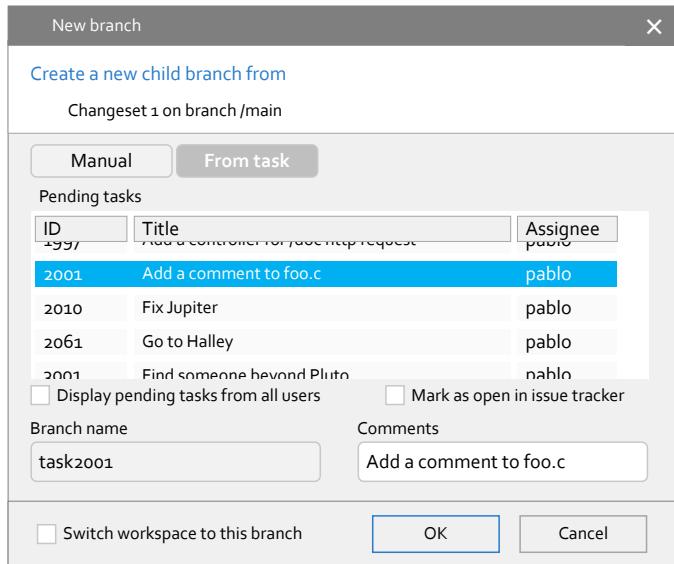
The dialog to create branches in the GUIs (Linux, macOS and Windows) looks like the following:



Where you can type the branch name and an optional comment. Branches can have comments in Plastic, which is something very useful because these comments can give extra info about the branch and also be rendered in the Branch Explorer.

As you see, there is checkbox to switch the workspace to the branch immediately after creation.

If you look carefully, you'll find there are two main options to create branches: manual and from task. The image above shows the manual option, and now we're going to see what the "from task" is all about.



You can connect Plastic to a variety of issue trackers and one of the advantages is that you can list your assigned tasks and create a branch directly from them with a single click. Learn more about how to integrate with issue trackers reading the [Issue Trackers Guide](https://www.plasticscm.com/download/help/) [<https://www.plasticscm.com/download/help/>]

taskandissuetrackers].

Diffing changesets and branches

The next key tool to master in Plastic is the diff. You'll spend probably more time diffing and reviewing changes than actually making checkins.

Diffing from the command line

Suppose you want to diff your first changeset; it would be as simple as using the `cm diff` command:

```
>cm diff cs:1  
A "foo.c"
```

It tells you that in changeset 1 you simply added `foo.c`.

Now, if you diff changeset 2:

```
>cm diff cs:2  
C "foo.c"
```

The command tells you that you actually modified `foo.c` on that changeset.

The `cm diff` command is the one responsible for diffing both changesets/branches and actual files.

Since the changes we made were pretty basic so far, the diffs will be very simple. So let me just show you a diff from a branch I have on one of my local repos:

```
>cm diff br:main/scm23606  
C "art\me\me.jpg"  
C "art\ironman\IronMan.jpg"  
C "code\plasticdrive\FileSystem.cs"  
M "code\cgame\FileSystem.cs" "code\plasticdrive\FileSystem.cs"
```

As you can see, the diff is richer this time, and it tells how I changed three files and moved one of them in the branch `main/scm23606`.

So far, we only diffed changesets and branches, but it is possible to diff files too. I'm going to print the revision ids of the files changed in `scm23606` to diff the contents of the files:

```
>cm diff br:main/scm23606 --format="\{status\} \{path\} \{baserevid\} \{revid\}"  
C "art\me\me.jpg" 28247 28257  
C "art\ironman\IronMan.jpg" 27600 28300  
C "code\plasticdrive\FileSystem.cs" 28216 28292  
M "code\plasticdrive\FileSystem.cs" -1 28292
```

And then, I can diff the change made in `FileSystem.cs` as follows:

```
>cm diff revid:28216 revid:28292
```

Instead of a text diff printed on your console, `cm diff` launches the diff tool configured for text files. Plastic comes with a built-in diff and merge tool we call Xdiff/Xmerge, and then a GUI like the following displays:

The screenshot shows a unified diff interface with two panes. The left pane is labeled "revid:28216" and the right pane is labeled "revid:28292". Both panes show code snippets with line numbers. A vertical dashed line separates the two panes. Changes are highlighted in yellow. The bottom of the window has buttons for "Visual diff", "Run text diff", and "Restart diff".

```
> revid:28216
15     ..... [-private string::MountPoint;
16     ..... [-private int::count_];
17     ..... [-private TreeContent::fileSystemContent=-null;
18     ..... [-private FileCache::mCache=-new FileCache();
19
20     ..... [-private MemoryStream::mSelectorBytes=-new MemoryStream();
21
22     ..... [-private FileHandles::mHandles=-new FileHandles();
23
24     ..... [-public SelectorFS(string::mountPoint,-string::clientconf,-string::sel
25
26     ..... [-public DeleteFile(string::path)[]..];
27
28     ..... [-public bool::DeleteFile(string::path)[]..];
29
30     ..... [-void writesSelector()[]..];
31
32     ..... [-public int::CreateDirectory(
33         ..... [-string::filename,
34         ..... [-DokanFileInfo::info]..];
35
36     ..... [-public int::OpenDirectories(
37         ..... [-string::filename,
38         ..... [-DokanFileInfo::info]
39
40     ..... [-log.DebugFormat("Open::Directory::{0}",filename);
41     ..... [-info.Context+= count_++];
42
43     ..... [-if (DirectoryExists(VirtualPath.getPath(filename))
44     ..... [-return 0];
45
46     ..... [-return -DokanNet.ERROR_PATH_NOT_FOUND--1;
47
48     ..... [-}
49
50     ..... [-]
51
52     ..... [-log.DebugFormat("Open::Directory::call::{0}",filename);
53     ..... [-info.Context+= count_++];
54
55     ..... [-if (!DirectoryExists(VirtualPath.getPath(filename))
56     ..... [-return 0;
57
58     ..... [-return -DokanNet.ERROR_PATH_NOT_FOUND--1;
59
60     ..... [-}
61
62     ..... [-]
63
```

```
revid:28292
17     ..... [-private string::MountPoint;
18     ..... [-private int::count_];
19     ..... [-private TreeContent::fileSystemContent=-null;
20     ..... [-private FileCache::mCache=-new FileCache();
21
22     ..... [-private MemoryStream::mSelectorBytes=-new MemoryStream();
23
24     ..... [-private FileHandles::mHandles=-new FileHandles();
25
26     ..... [-public SelectorFS(string::mountPoint,-string::clientconf,-string::sel
27
28     ..... [-public DeleteFile(string::path)[]..];
29
30     ..... [-public bool::DeleteFile(string::path)[]..];
31
32     ..... [-void writesSelector()[]..];
33
34     ..... [-public int::CreateDirectory(
35         ..... [-string::filename,
36         ..... [-DokanFileInfo::info]..];
37
38     ..... [-public int::OpenDirectories(
39         ..... [-string::filename,
40         ..... [-DokanFileInfo::info]
41
42     ..... [-log.DebugFormat("Open::Directory::{0}",filename);
43     ..... [-info.Context+= count_++];
44
45     ..... [-if (DirectoryExists(VirtualPath.getPath(filename))
46     ..... [-return 0];
47
48     ..... [-return -DokanNet.ERROR_PATH_NOT_FOUND--1;
49
50     ..... [-}
51
52     ..... [-log.DebugFormat("Open::Directory::call::{0}",filename);
53     ..... [-info.Context+= count_++];
54
55     ..... [-if (!DirectoryExists(VirtualPath.getPath(filename))
56     ..... [-return 0;
57
58     ..... [-return -DokanNet.ERROR_PATH_NOT_FOUND--1;
59
60     ..... [-}
61
62     ..... [-]
63
```

We think GUIs are better tools to show diffs than the typical unified diff displayed by many tools, and that's why we launch a GUI even when the diff is invoked from the command line.

In case you really want to see a unified diff printed on console (because you need to run the diff from a terminal without GUI support, for instance), [check this](https://www.plasticscm.com/download/help/unixdifffromconsole) [<https://www.plasticscm.com/download/help/unixdifffromconsole>].

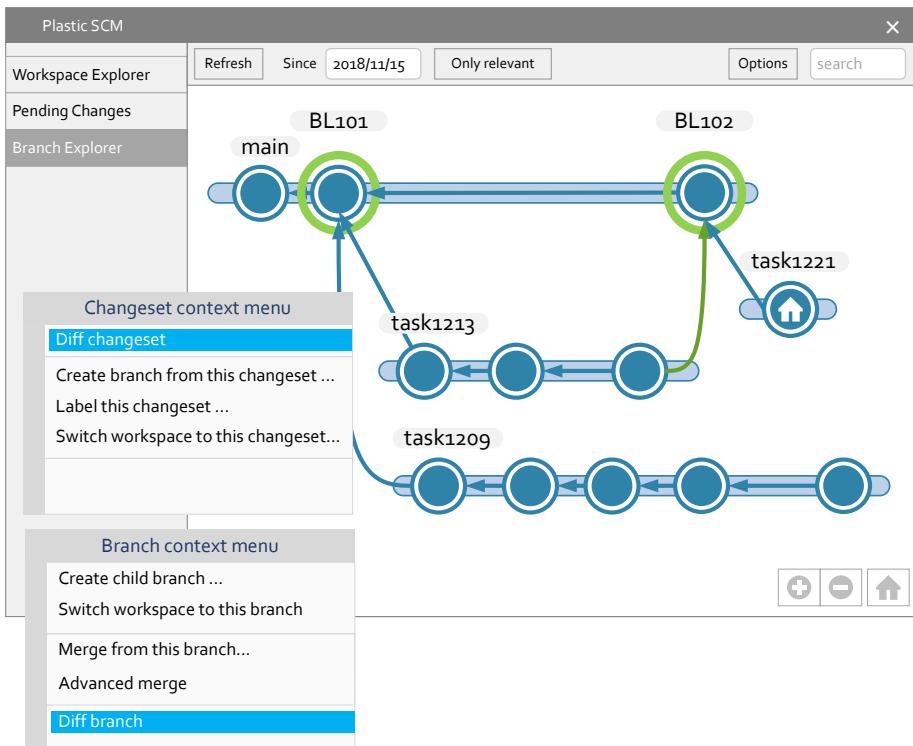
Explore the `cm diff help` to find all the available options. You can diff branches, individual files, changesets, labels (which is the equivalent to diffing the changeset), or any pair of changesets you want to see the changes between them.

```
>cm diff help
```

Differing from the GUIs

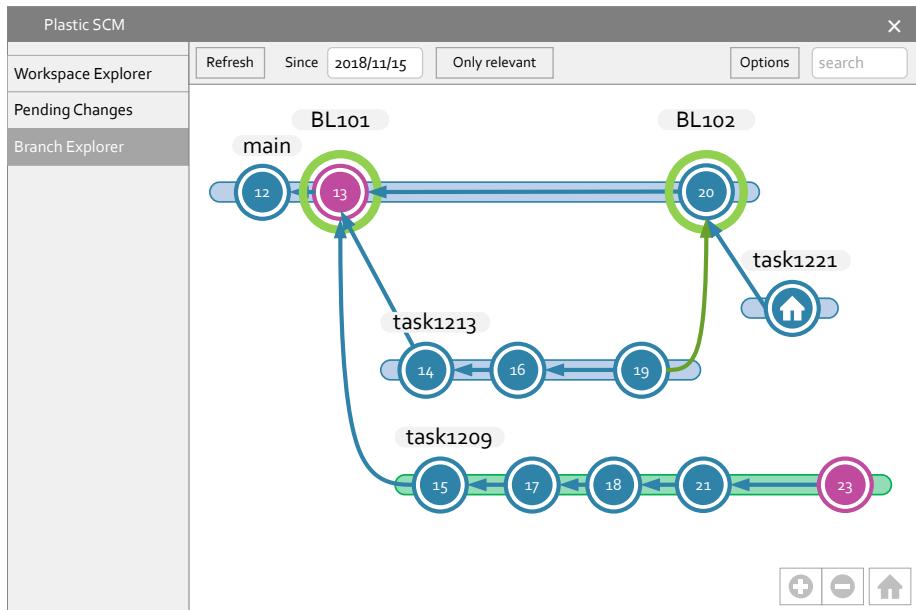
Well, it is no secret that when it comes to differencing, we put much more attention into GUIs than command line. This is because we think that 99% of the time developers will difference using a Plastic GUI instead of the command line.

Differing changesets and branches is very simple from the Branch Explorer; examine the following figure, where you'll see how there are difference options in the context menus of branches and changesets (and labels).



Differing an entire branch is as simple as right clicking on it to select it (you'll see it changes color) and then selecting the "diff branch" option.

As the following figure shows, differing a branch is the equivalent of differing its head changeset and its parent changeset.



In the figure, you can see how diffing branch **task1209** is equivalent to diffing changesets **13** and **23**.

In fact, you can see the selected changesets are colored in violet; when you multi-select changesets they are colored differently to highlight the fact that many csets are selected. And, you can easily right click and run "diff selected".

The diffs of changesets and branches in the GUI are very powerful GUIs like the following mockup shows:

Plastic SCM – diff changeset xxx

Modified some integration code and the render

<< < 1/3 > >> filter

Path	Status	Size	Type	Date modified
- C Changed				
src/core/Integration.cs	Checkout	3 KB	Text	3 min ago
src/graphics/Render.cs	Changed	1 KB	Text	5 min ago
- D Deleted				
FS/FileSystem.h	Removed	3 KB	Text	10 min ago

```

29 .....}
30
31 .....public bool DeleteFile(string path)
32 {
33 .....//·modified
34 .....return false;
35 .....}
36
37 .....void WriteSelector()
38 {
39 .....//·write-selector·function
40 .....}
41
42 .....M public int CreateDirectory(
43 .....string fileName,
44 .....DokanFileInfo info)
45 {
46 .....return 0;
47 .....}
48

```

29}

```

30
31 .....M public int CreateDirectory(
32 .....string fileName,
33 .....DokanFileInfo info)
34 {
35 .....return 0;
36 .....}
37
38 .....public bool DeleteFile(string path)
39 {
40 .....//·modified
41 .....return false;
42 .....}
43
44 .....void WriteSelector()
45 {
46 .....//·write-selector·function
47 .....}
48

```

The changeset/branch diff window first summarizes the changes on a tree with changed, deleted, added and moved, and then shows the actual side-by-side diff of the selected file.

One of the great advantages of the built-in diff in Plastic is that it comes with the ability to do semantic diffs. This means that for certain languages like C#, C++, C and Java (and then a number of community-contributed parsers) the diff is capable of tracking moved fragments of code. In the figure above, you can see how the `CreateDirectory` method was identified as moved to a different location.

Learn more about what "[semantic version control](#)" [<https://www.plasticscm.com/features/semantic-version-control>] can do.

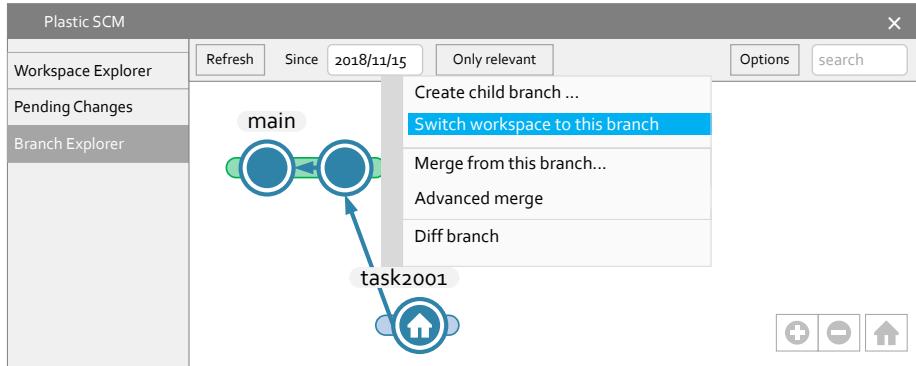
Merge a branch

Let's rewind to where we left our very simple example where we created branch `task2001` and made a checkin on it. What if we now want to merge it back to main?

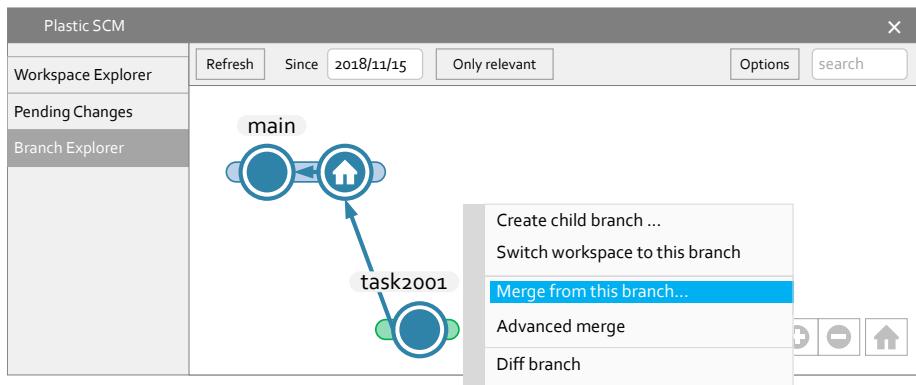
Merging is very simple; you switch your workspace to the destination branch and then merge **from** the source branch or changeset. This is the most typical merge. There is another option named "merge to" which doesn't even require a workspace at all to run it, but we'll start with the most common one.

Merging from the GUI

This is what our branch explorer looks like at this point with branches main and `task2001`. If you remember, the workspace was in `task2001`, so the first thing we'll do is to switch to main, which is as simple as selecting the branch, right clicking on it, and running the "Switch workspace to this branch" action.



Once the switch is complete, the Branch Explorer will look as follows and we'll right click on `task2001` to run a "merge from this branch":



The GUI will guide us through the merge process. In this case, there's no possible conflict since there are no new changesets on main that could possibly conflict with `task2001`, so all we have to do is to checkin the result of the merge. In the Pending Changes view, you'll see there is a "merge link" together with the modified `foo.c`. Something as follows:

Plastic SCM

Workspace Explorer Pending Changes Branch Explorer

Type your comments here

Path	Status	Size	Type	Date modified	
- C foo.c	Changed	Replaced/Checkedout	3 bytes	Text	1 min ago

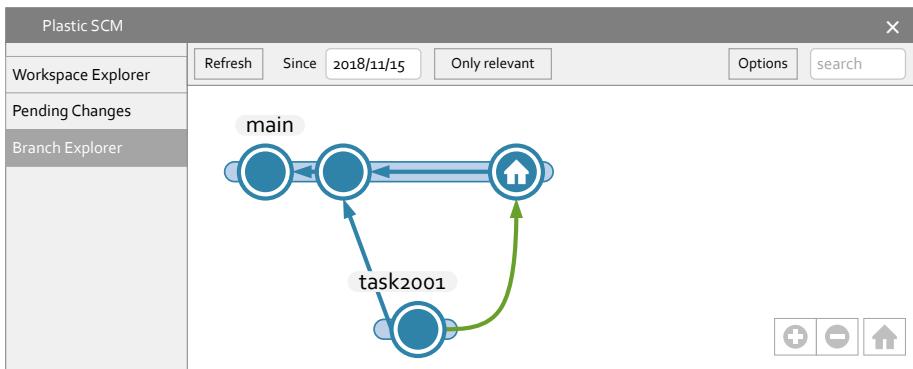
1 pending merge link(s). Pending merge links will be committed when you checkin your changes

Merge link description

Merge from cs:2 at main/task2001@default@localhost:8087

See "[Plastic merge terminology](#)" in the merge chapter to learn what "replaced" means, and also the other possible statuses during the merge.

Now, once the checkin is done, the Branch Explorer will reflect that the branch was correctly merged to main as follows:



We performed the entire merge operation from the Branch Explorer, but you can also achieve the same from the Branches view (list of branches) if you prefer.

Merging from the command line

By the way, it is also possible to merge from the command line. I'm going to repeat the same steps explained for the GUI but this time running commands.

First, let's switch back to main:

```
>cm switch main
Performing switch operation...
Searching for changed items in the workspace...
Setting the new selector...
Plastic is updating your workspace. Wait a moment, please...
Downloading file c:\Users\pablo\wkspaces\demowk\foo.c (8 bytes) from default@localhost:8087
Downloaded c:\Users\pablo\wkspaces\demowk\foo.c from default@localhost:8087
```

And now, let's run the merge command:

```
>cm merge main/task2001  
The file /foo.c#cs:2 was modified on source and will replace the destination version
```

The `cm merge` command simply does a "dry run" and explains the potential merge conflicts.

To actually perform the merge we rerun with the `--merge` flag:

```
> cm merge main/task2001 --merge  
The file /foo.c#cs:2 was modified on source and will replace the destination version  
Merging c:\Users\pablo\wkspaces\demowk\foo.c  
The revision c:\Users\pablo\wkspaces\demowk\foo.c@cs:2 has been loaded
```

Let's check the status at this point:

```
>cm status  
/main@default@localhost:8087 (cs:1 - head)  
  
Pending merge links  
    Merge from cs:2 at /main/task2001@default@localhost:8087  
  
Changed  
    Status          Size      Last Modified     Path  
    Replaced (Merge from 2)   21 bytes   14 seconds ago   foo.c  
  
Added  
    Status          Size      Last Modified     Path  
    Private        6 bytes   7 minutes ago    bar.c
```

As you can see, it says `foo.c` has been "replaced", same as the GUI did.

A checkin will confirm the merge:

```
>cm ci -c "merged from task2001"  
The selected items are about to be checked in. Please wait ...  
\ Checkin finished 0 bytes/0 bytes [#####] 100 %  
Modified c:\Users\pablo\wkspaces\demowk  
Replaced c:\Users\pablo\wkspaces\demowk\foo.c  
Created changeset cs:3@br:/main@default@localhost:8087 (mount:'/')
```

And this way we completed a full branch merge from the command line ☺

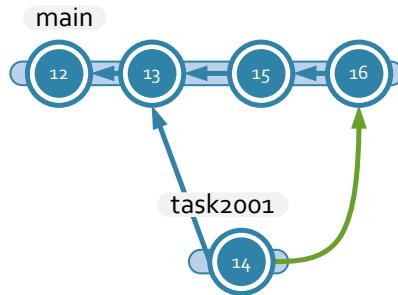
Learn more

This was just an intro to show you how to perform a basic merge, but there is a full chapter in this guide about the [merge operation](#) where you'll learn all the details required to become an expert.

Annotate/blame a file

Annotate, also known as blame, tells you where each line of a file comes from. It is extremely useful to locate when a given change was made.

Consider the following scenario where a file `foo.cs` was modified in each changeset:



And now consider the following changes made to the file in each changeset:

pablo	borja	pablo	jesus
11 10 class Program 20 { 30 static void Main() 40 { 50 Print("Hello"); 60 for (int i=0; i< 10; ++i) 70 Print(i); 80 } 90 }	33 10 class Program 20 { 30 static int Main() 40 { 50 Print("Hello"); 60 for (int i=0; i< 10; ++i) 70 Print(i); 80 } 90 }	15 10 class Program 20 { 30 static int Main() 40 { 50 Print("Bye"); 60 for (int i=0; i< 13; ++i) 70 Print(i); 80 } 90 }	16 10 class Program 20 { 30 static int Main() 40 { 50 Print("Bye"); 60 for (int i=0; i< 16; ++i) 70 Print(i+1); 80 } 90 }

miguel

14 10 class Program 20 { 30 static int Main() 40 { 50 Print("Hello"); 60 for (int i=0; i< 11; ++i) 70 Print(i+1); 80 } 90 }
--

The code snippets show the evolution of the `foo.cs` file across four authors (pablo, borja, pablo, jesus) and one additional row for miguel. The first three authors (pablo, borja, pablo) have identical code. The fourth author (jesus) has a different codebase. The fifth row (miguel) shows a partial view of the file. Colored highlights indicate changes: green highlights the addition of the `Print(i+1);` line in revision 16, and red highlights the modification of the loop condition from `i< 10` to `i< 16`. The revision numbers (11, 12, 13, 14, 15, 16) are circled in green above their respective code blocks.

I marked in green the changes made in each of the revisions compared to the previous one, and also specified the author of each change in the top of the file.

The annotate/blame of the file in revision 16 will be as follows:

			16
pablo	cs:12 br:/main	9 days ago	10 class Program
pablo	cs:12 br:/main	9 days ago	20 {
borja	cs:13 br:/main	3 days ago	30 static int Main()
pablo	cs:12 br:/main	9 days ago	40 {
pablo	cs:15 br:/main	1.5 days ago	50 Print("Bye");
jesus	cs:16 br:/main	45 min ago	60 for (int i=0; i< 16; ++i)
miguel	cs:14 br:/main/task2001	2 days ago	70 Print(i+1);
pablo	cs:12 br:/main	9 days ago	80 }
pablo	cs:12 br:/main	9 days ago	90 }

Please take some time to examine the example carefully until you really understand how it works. What annotate does is to take the contents of the file in the changeset 16, then walk back the branch diagram diffing the file until the origin of each line is found.

In our case, it is clear that lines 10, 20, 40, 80 and 90 come from the changeset 12 where the file was originally added.

Then line 30, where Borja changed the return type of the `Main` function from `void` to `int`, belongs to changeset 13.

Line 50 is where pablo changed `Print("Hello");` by `Print("Bye")` and it is attributed to changeset 15.

Then line 60 is marked in red because it was modified during a merge conflict; changeset 14 in `main/task2001` modified the line differently than changeset 15 in `main`, so Jesus had to solve the merge conflict and decided to modify the loop so it finally goes from 1 to 16.

Finally, line 70 comes from changeset 14 in `task2001` where Miguel modified the line to be `Print(i+1);`

Since Plastic always keeps branches (check "[We don't delete task branches](#)") the annotate tells you which was the task where a given change was introduced, which is of great help to link to the right Jira or any issue tracker you might use.

Please note that the GUI will use a gradient to differentiate the most recent changes from the older ones.

Annotate/blame from command line and GUI

Annotating a file is very easy from the GUI; just go to the Workspace Explorer, locate the file, right click on it and select "annotate".

From the command line it is not difficult either:

```
> cm annotate foo.c
```

You annotate a given revision of a file

You don't annotate a file but a given revision of a file. What I mean by this is that, in the previous example, it is not the same to annotate the file in changeset 16 than 14 or 15.

If you annotate 12, for example, all the lines will be attributed to changeset 12 since this was where the file was added. If you annotate 13, all lines will belong to 12 except line 30 that will be marked as created in 13.

If a line is modified several times, the line closer to the destination changeset will be the one marked as "creator" of the line.

Semantic annotate

You probably know by now that we are somehow obsessed with improving diffs and merges by parsing code. Our plan is to improve annotations also in such a way that if you move a fragment of code or a full function/method to a different location within the file, annotate considers that move and continues tracking back the line until it reaches its original creation point.

And, the goal is to extend that to cross-file operation too. So, expect great improvements in this area in the coming versions ☺

Push and pull

If you ever heard of distributed version control (DVCS), and you probably have unless you have been off of the planet in the last 10 years, then you know that the two key operations are push and pull.

I'm not going to explain what they mean in detail, since we have a chapter dedicated to distributed operation, but just introduce the basics.

Suppose you create a new repository and want to replicate the main branch we just created in the default repo to it.

```
> cm repository create cloned@localhost:8087  
> cm push main@default@localhost:8087 cloned@localhost:8087
```

The `cm push` command pushes the `main@default` branch to the "cloned" repo.

Imagine that, later, some new changes are created in `main@cloned` and you want to pull them to `main@default`.

```
> cm pull main@cloned@localhost:8087 default
```

I used a couple of repos on my localhost server, but the following would be possible:

```
> cm push main@default@localhost:8087 default@codice@cloud  
> cm push main@default@localhost:8087 newrepo@skull.codicefactory.com:9095
```

Provided that the destination repos `default@codice@cloud` and `newrepo@skull.codicefactory.com:9095`

were previously created.

Pushing and pulling from the GUI is even easier since there are options to do that from every branch context menu in the Branch Explorer view and also in the Branches view.

Finally, the GUI comes with a sync view to let you synchronize many branches in bulk. It is worthwhile to spend some time learning how the sync view works by reading the GUI [reference guide](#) [<https://www.plasticscm.com/download/help/syncview>].

Learn more

- Master the most frequent command line actions by reading the [command line guide](#) [<https://www.plasticscm.com/download/help/commandline>].

Keep reading this guide to learn more details about Plastic SCM ☺

ONE TASK - ONE BRANCH

Before we jump into how to do things in Plastic, I'm going to describe in detail the work strategy we recommend to most teams. Once that is clear, finding your way through Plastic will be straightforward.

As we saw in the section "[A perfect workflow](#)", task branches are all about moving high-quality work quickly into production.



Each task will go through a strict process of code development, code review and testing before being safely deployed to production.

This chapter covers how to implement a successful branch per task strategy, including all the techniques and tactics we learned over the years.

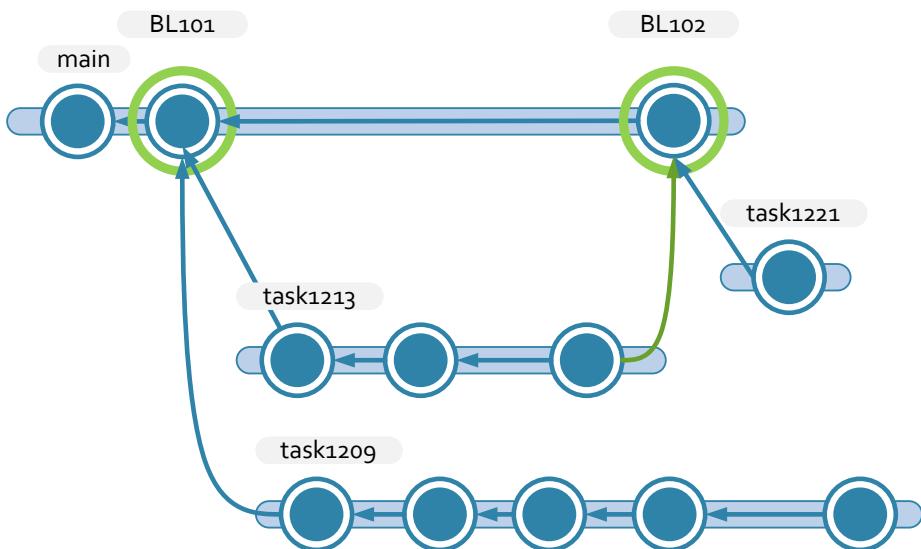
Branch per task pattern

We call it *task branches*, but the actual branch per task strategy [has been around for +20 years](#) [<http://www.bradapp.com/acme/branching/branch-creation.html#BranchPerTask>].

There are many different branching patterns but, over the years, we concluded task branches are really the way to go for 95% of the scenarios and teams out there.

The pattern is super simple; for each new task in your issue tracker, you create a new branch to work on.

The following is a typical diagram (it perfectly fits with the Branch Explorer visualization in Plastic) of a branch per task pattern:



You see how `task1213` is already finished and merged back to `main`, `task1209` is longer and still ongoing (and it has many changesets to it), and then `task1221` was just created and the developer only performed a single checkin on it.

All about branching patterns

If you want to master branching patterns, I strongly recommend that you grab a copy of "[Software Configuration Management Patterns: Effective Teamwork, Practical Integration](#)" [<http://scmpatterns.com>] by Steve Berczuk with Brad Appleton.

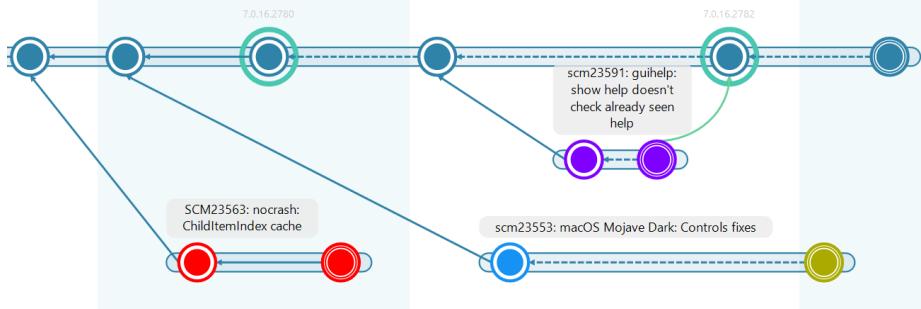
It might be close to a couple of decades old already, but it is still relevant since it is the real bible on branching.

In fact, it was part of the inspiration for why we created Plastic SCM in the first place ☺

Branch naming convention

We like to stick to the following pattern: prefix + task number. That's why the branches in the example above were named `task1213`, `task1209` and `task1221`. "task" is the prefix and the number the actual task number in the associated issue tracker.

The following is an actual screenshot from our repo:



You can see how we use the prefix "SCM" and then the task number. In our case, we use SCM because we build a *source code management* system ☺. You see how some team members prefer SCM in uppercase and others use lowercase... well, it is just a matter of self-expression I believe.



We also use the prefixes to link with the issue tracker, for example, with [Jira](https://www.plasticscm.com/download/help/jiraintegrationclientconfiguration) [<https://www.plasticscm.com/download/help/jiraintegrationclientconfiguration>].

You also see how the screenshot shows a description for each branch together with the number. The Branch Explorer retrieves that from the issue tracker or alternative the branch description when you select the option "[display branch task info](#)" [<https://www.plasticscm.com/download/help/displaybranchtaskinfo>].

We see some teams using long branch names, where the name actually tells what the task is about. Is that wrong? Not really, but we truly prefer to direct the conversation based on univocal numbers. In fact, see what our Kanban board looks like on any day: the entire conversation is always about tasks, and there's no doubt where the associated code is.

Implement (13)

Open	Locked	Done
23552: macOS Mojave: Dialogs layout huge button issue   	(locked reason: Waiting for Peter actions) 22393: Semantic 2.0: manual test periparser 0.0 version 	6 23520: mcga: mv - apply local moves   
23559: nocra sh: study mutex  	23477: Gluon:move percentage set to 40 error out of range  	23463: macos: Standalone app to repro frozen Cloud update   

Task branches are short

Remember that Scrum rule that says tasks shouldn't be longer than 16 hours to ensure they are not delayed forever and to keep the project under control? Well, we love that.

At the time of writing, we had just switched to Kanban for a couple of months after almost 300 sprints of two weeks each. We changed to Kanban to reduce task cycle times, but we like many practices of Scrum. And keeping tasks short is one of them.

Task branches must be closed quickly. It is great to have many small tasks you can close in just a few hours each. It is good to keep the project rhythm, to keep the wheels spinning and never stop deploying new things. A task that spans for a week simply stops the cycle.

"But some tasks are long" – I hear you crying. Sure, they are. Splitting bigger tasks into smaller ones is an art and science in itself, and something you need to master to really get the best out of task branches and a solid overall continuous delivery cycle.

There are a few red lines to keep in mind:

- **Don't create "machete cut" tasks.** You ask someone to cut a task into smaller pieces and then they go and create a pile of smaller ones that don't make any sense in isolation and can't be deployed independently. No, that's not what we are looking for. Of course, creating shorter tasks from a bigger one can be daunting at times, but nothing you can't solve throwing a little bit of brain power to it.
- **Every team member needs to buy it.** It is not just about "let's make all tasks consistently shorter", it is about understanding life is easier if you can close and deliver a piece of a larger assignment today instead of struggling with it for a few days before delivering anything. It is all about openly communicating progress, avoiding silos, and removing "I'm at 80% of this" day after day. Predictability, responding to problem fast, and the entire agile philosophy is under the "short branches" motto.

Task branches are not feature branches

I admit sometimes I wrongly used both terms interchangeably. Feature branches got very popular thanks to Git Flow in the early Git explosion days. Everyone seemed to know what a feature branch was, so sometimes I used it as a synonym for task branches. They are not.

A feature can be much longer than a task. It can take many days, weeks or months to finish a feature. And it is definitely not a good idea to keep working on a branch for that long unless you want to end up with a [big-bang integration](#).

I'm not talking just about merge problems (most of our reality here at Plastic is about making impossible merges easy to solve, so that's not the problem), but about project problems. Delay a merge for a month and you'll spend tons of unexpected time making it work together with the rest of the code, and solving merge problems will be the least of your problems.

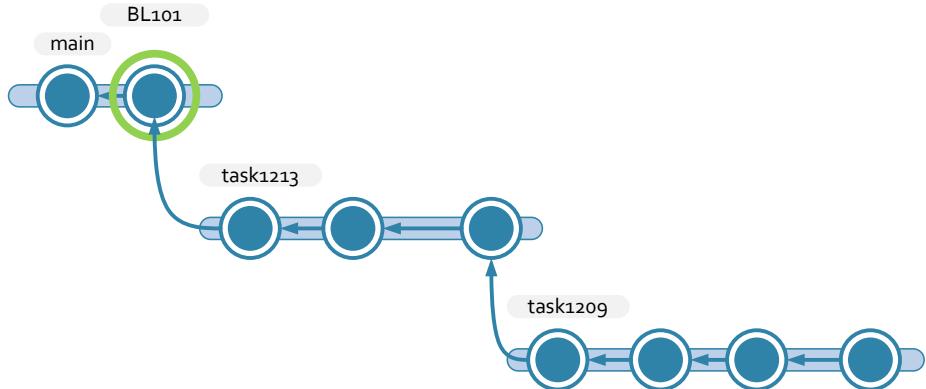
Believe me, continuous integration, task branches, shorter release cycles were all created to avoid *big-bangs*, so don't fall into the trap.

To implement a feature, you will create many different task branches, each of them will be merged back as soon as it is finished, and one day the overall feature will be complete. And, before you ask: use [feature toggles](#) if the work is not yet ready to be public, but don't delay integrations.

Keep task branches independent

Here is the other skill to master after *keep task branches short*: keep task branches independent.

Here is a typical situation:



You just finished `task1213` and you have to continue working on the project, so the obvious thing to do is to simply continue where you left it, right?

Wrong.

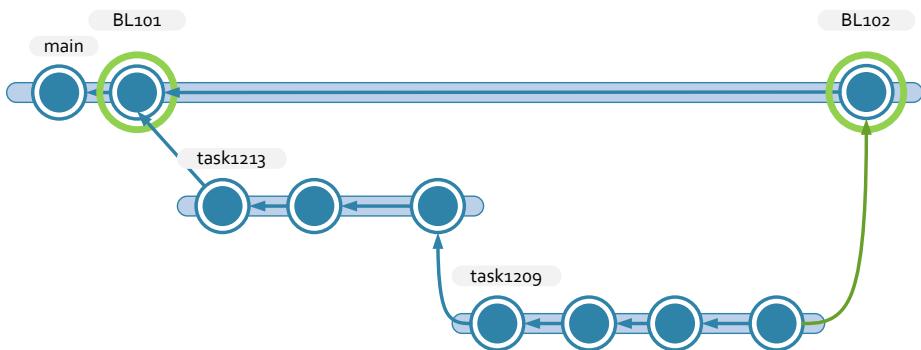
This happens very often to new practitioners, they just switched from doing checkins to trunk continuously and they feel the urge to use their previous code, and even a bit of vertigo (merge-paranoia) if they don't.

The question you have to ask yourself (or your team mates) twice is: do you really need the code you just finished in `task1213` to start `task1209`? Really? Really?

Quite often, the answer will be no. Believe me. Tasks tend to be much more independent than you first think. Yes, maybe they are exactly on the same topic, but you don't need to touch exactly the same code, you can simply add something new, and trust the merge will do its job.

There is a scenario where all this is even clearer and more dangerous: suppose that `1213` and `1209` are bugfixes instead of tasks. You don't want one to depend on the other, you want them to hit main and be released as quickly as possible, even if they touch the same code, they are different fixes. Keep them independent!

And now you also get the point of why keeping tasks independent matters so much. Look at the following diagram:



`task1209` was merged *before* `task1213`. Why is this a problem? Well, suppose each task goes through a review and validation process:

- `task1209` was reviewed and validated, so the CI system takes it and merges it.
- But, `task1213` is still waiting. It wasn't reviewed yet... but it reached main indirectly through `1209`.

See the problem?

Well, you can say: yes, but all you have to do is to tell the team that `1209` doesn't have to be merged before `1213` is ready.

Yes, I hear you. And it worked moderately well in the times of manual integration, but with CI systems automating the process now, you are just creating trouble. Now, you have to manually mark the task somehow so that it is not eligible for merge (with our mergebots and CI integrations, it would be just a matter of setting an attribute to the branch). You are just introducing complexity and slowing the entire process down. Modern software development is all about creating a continuous flow of high-quality changes. These exceptions don't help.

What if you really need tasks to depend on each other

There will be cases where dependencies will be *really* needed. Simply ask yourself three times before taking the dependency for granted.

If, after all, tasks need to depend on each other, then you need to control the order in which they'll be integrated. If you are triggering the merges manually (discouraged) then simply ensure there is a note for the *build-master* or *integrator* in charge.

If you have an automatic process in place, you'll need to mark branches somehow so they are not eligible for merge by the CI system or the mergebot.

Techniques to keep branches independent



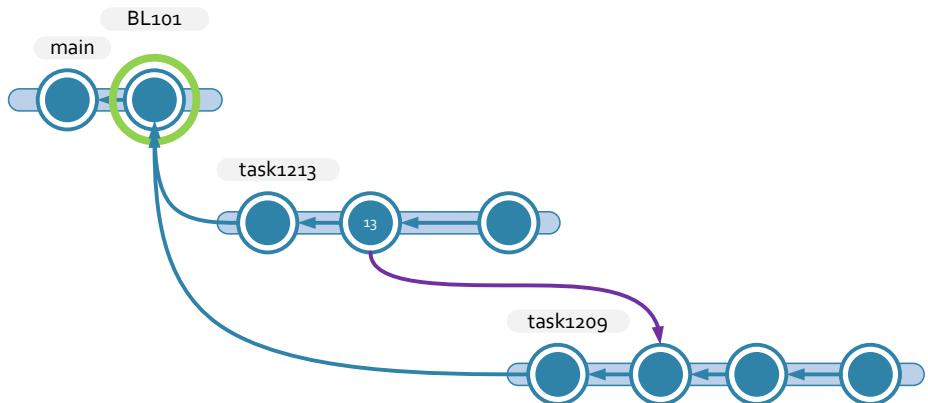
This is a pretty advanced topic, so if you are just finding your way through Plastic, maybe it is better if you simply come back to it when you have mastered the basics.

There will be other cases where, with a little effort, you'll be able to keep tasks independent.

Very often when you need to depend on a previous task, it is because you need just *some* of the changes done there.

Use cherry-pick carefully to keep branch independent

If that's the case, and provided you were careful enough to isolate those changes in an isolated changeset (more on that in the "[Checkin often and keep reviewers in mind](#)" section) you can do a cherry-pick merge (as I explain in the "[Cherry picking](#)" section).



A cherry-pick will only bring the changes made in changeset 13, not the entire branch, and the branches will continue being mostly independent and much safer to merge independently than before.

Of course, this means you must be careful with checkins, but that's something we also strongly encourage.

Add a file twice to keep tasks independent

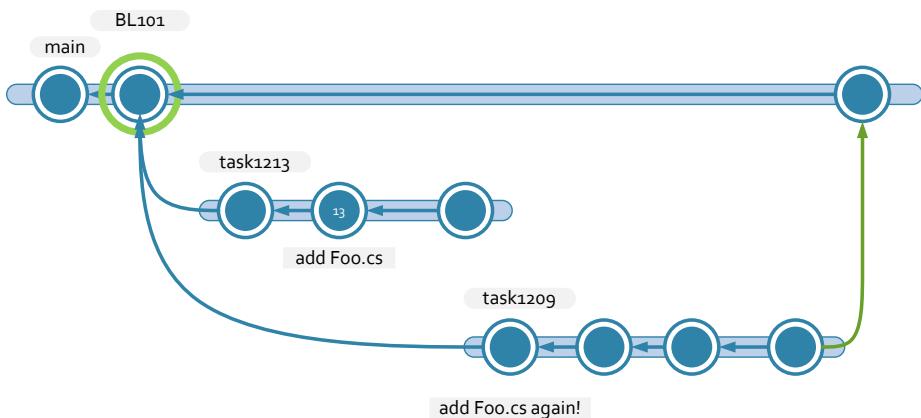
There is a second technique that can help when you add a new file on `task1213` and you really need this file in the next task. It can be some code you really need to invoke. This is a little bit contrived, but it works: simply add the file again in `task1209`. Of course, this only works when you just need something *small*, if you need everything, copying is definitely not an option.

Now, you think I went crazy: copy a file manually and checkin again? Isn't it the original sin that version control solves?

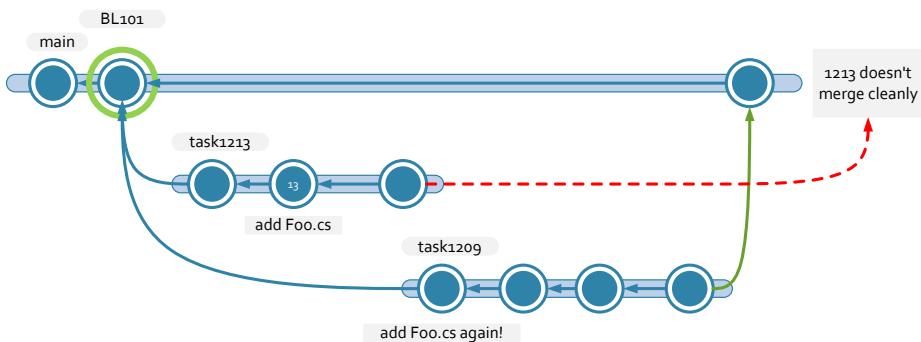
Remember, I'm trying to give you choices to keep tasks independent, which is something quite desirable. And yes, this time, this technique can help.

You'll be creating a merge conflict for the second branch (an [evil twin](#) as we call it) that you'll need to solve, but tasks will stay independent.

The sequence will be as follows: `task1209` will merge cleanly.



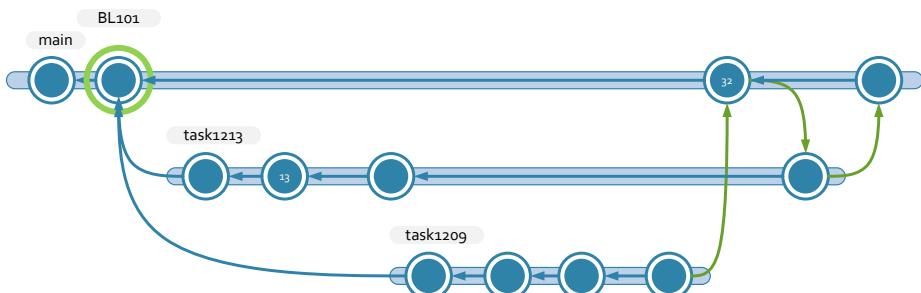
Then, when **task1213** tries to merge, a conflict is raised:



If you are doing manual merges to main, all you have to do is to solve the conflict and checkin the result. It won't be that hard.

But, if you are using a CI system or a mergebot in charge of doing the merges, there won't be a chance to do manual conflict resolution there, so the task will be rejected.

If that happens, you'll have to follow what we call a *rebase cycle*: simply merge down from main to **task1213** to solve the problem and then the branch will merge up cleanly.



Note for Git users

Git rebase is very different than what we call rebase in Plastic. Rebase in Plastic is simply "merge down" or "changing the base". Branch **task1213** "started from" **BL101** before, but after the rebase it virtually starts from changeset **32**.

Checkin often and keep reviewers in mind

Ever heard this? **Programs must be written for people to read, and only incidentally for machines to execute.** It is from the legendary book "Structure and Interpretation of Computer Programs (MIT)".

It is a game changer. We should write code thinking about people, not computers. That's the cornerstone of readable code, the key of simple design. It is one of the cornerstones of agile, extreme programming and other techniques that put code in the center of the picture.

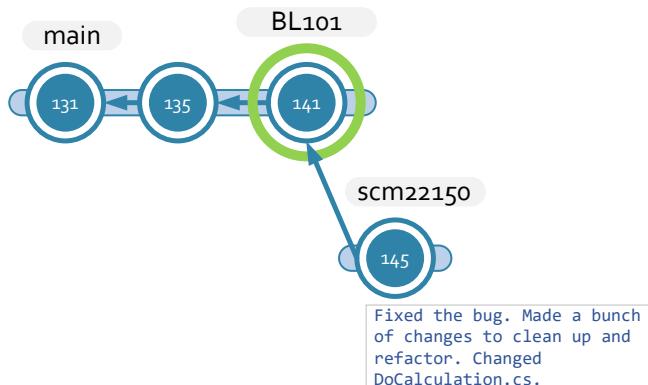
Well, what if you also checkin for people to read? Yes, it might not be easy at first, but it really pays off.

Let's start with a few anti-patterns that will help me explain my point here.

Antipattern 1: Checkin only once

Imagine that you have to do an important bug fix or add a new feature, but before that, you need to clean up some code.

This is one way to do it:

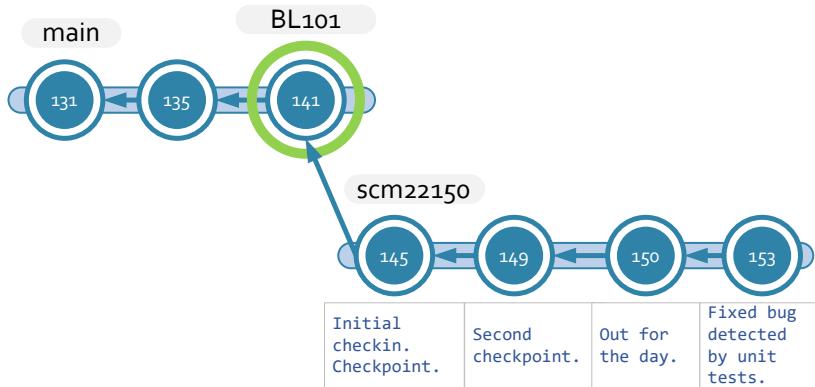


Then the reviewer comes, diffs the changeset number 145, finds there are 100 modified files and... goes for a coffee, or lunch, or wants to leave for the day... ouch!

And this is how the traditional way to diff branches provokes context switches, productivity loss or simply "ok, whatever, let's approve it".

Antipattern 2: Checkin for yourself

Let's try again with a different checkin approach:



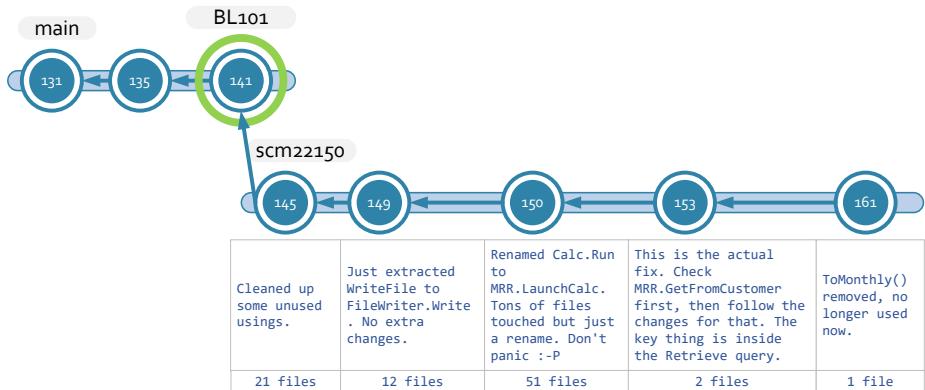
This time the developer checked in several times during development. It was probably very helpful for him because he protected his changes to avoid losing them in the event of a weird crash or something. Multiple checkins can also help when you are debugging, or doing performance testing, because instead of commenting code out and going back and going crazy, you create real "checkpoints" you know you can go back later safely if you get lost in the way.

But the reviewer will go and simply diff the entire branch because the individual checkins won't be relevant for him. And the result will be as demotivating as it was in the previous case: 100+ files to review. Ouch! Go for coffee.

Checkin for the reviewer

Now, let's follow the rule we all use here at Códice: checkin with the reviewer in mind. Every single checkin has to help someone else following your train of thoughts, follow your steps to really understand what happened.

Let's try again:



Now, as a reviewer, you won't go and diff the entire branch. You'll diff changeset by changeset. And, you'll be simply following the pre-recorded explanation the author made to clarify each step of the task. You won't have to find yourself against a bold list of 100+ modified files. You'll go step by step.

First, you see 21 files modified, but the comment says it was just about cleaning up some C# usings. The list of 21 files is not terrifying anymore, it is just about removing some easy stuff. You can quickly glance

through the files or even skip some of them.

Then, the next 12 files are just about a method extracted to a new class plus the affected callers having to adapt to the new call format. Not an issue either.

Next, comes 51 files, but the comment clearly says it is just because of a method was renamed. Your colleague is clearly telling you it is a trivial change, probably done thanks to the IDE refactoring capabilities in just a few seconds.

Then, it comes the real actual change, the difficult one. Fortunately, it only affects 2 files. You can still spend quite a lot of time on this, really understanding why the change was done and how it works now. But it is just two files. Nothing to do with the setback produced by the initial vision of 100 files changed.

Finally, the last change is a method that has been removed because it is no longer invoked.

Easier, isn't it?

Objection: But... you need to be very careful with checkins, that's a lot of work!

Yes, I hear you. Isn't it a lot of work to write readable code? Isn't it easier and faster to just put things together and never look back? It is not, right? The extra work of writing clean code pays off in the mid-term when someone has to touch it, to modify it. It pays off even if the "other person touching the code" is your future self coming back just three months later.

Well, the same applies to carefully crafted checkins. Yes, surely, they are harder than "checkpoint" checkins, but they really (really!) pay off.

And, the best thing is that you get used to working this way. Same as writing readable code becomes a part of your way of working once you get used to it, doing checkins with reviewers in mind shortly stop consuming extra brain-cpu cycles. It just becomes natural.

And, since development is a team sport, you'll benefit from others doing that too, so everything simply flows.

Task branches turn plans into solid artifacts that developers can touch

This is where everything comes together. As you can see, branches don't live in isolation. They are not just *version control artifacts* that live on a different dimension than the project. It is not like *branches are for coders* and plans, scrums, tasks and all that stay on a different galaxy. Task branches close the gap.

Very often, developers not yet in branch-per-task tend to think of tasks as some weird abstract artifact separated from the real job. It might sound weird, but I bet you experienced that some time. When they start creating a branch for each task, tasks become something they touch. Tasks become their reality, not just a place where some crazy pointy-haired-boss asked them to do something. Task branches change everything.

Handling objections

But, do you need a task in Jira/whatever for every task? That's a lot of bureaucracy!

It is a matter of getting used to. And it takes no time, believe me. We have gone through this many times already.

Once you get used to it, you feel naked if you don't have a task for the work you are doing.

Entering a task will probably be part of some product manager/scrum master/you-name-it most of the times, but even if, as a developer, you have to enter it (we do it very often), remember every minute saved in describing what to do will save you time multiplied by 10 in the form of questions, problems and misunderstandings.

I know this is obvious for many teams and you don't need to be convinced about how good having an issue tracker for everything is, but I consider it appropriate to enforce the key importance of having *tasks in branch per task*.

We use an issue tracker for bugs, but not new features.

Well, use it for everything. Every single change you do in code must have an associated task.

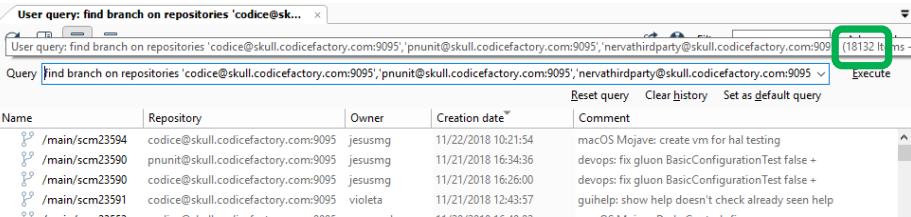
But some new features are big and they are just not one task and...

Sure, that's why you need to split them. Probably what you are describing is more a *story* or even an *epic* with many associated tasks. I'm just focusing on tasks, the actual units of work. The smallest possible pieces of work that is delivered as a unit.

Well, fine, but this means thousands of branches in version control. That's unmanageable!

No, it is not. Plastic can handle thousands of branches, no problem with that.

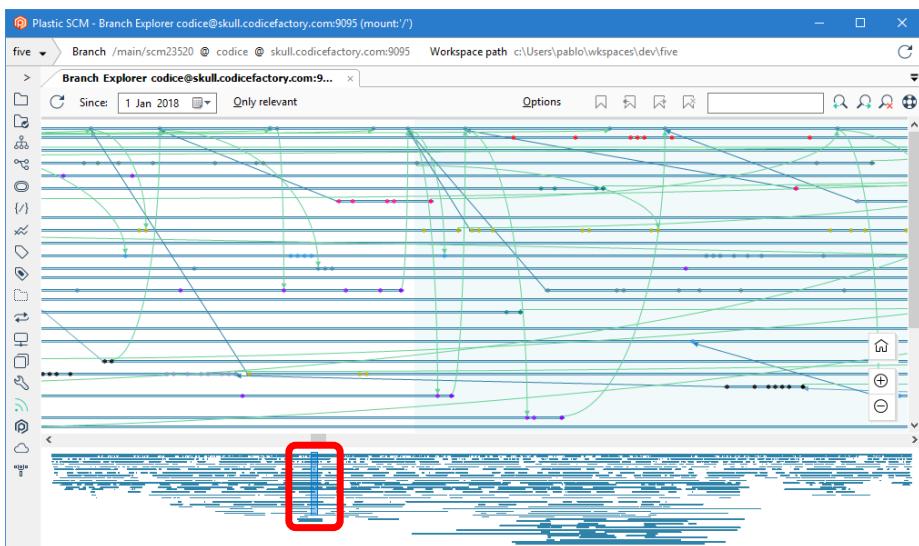
Look, this is the list of branches of our repositories that handle the Plastic source code (sort of recursive, I know 😊):



Name	Repository	Owner	Creation date	Comment
git /main/scm23594	codice@skull.codicefactory.com:9095	jesusmg	11/22/2018 10:21:54	macOS Mojave: create vm for hal testing
git /main/scm23590	punit@skull.codicefactory.com:9095	jesusmg	11/21/2018 16:34:36	devops: fix gluon BasicConfigurationTest false +
git /main/scm23590	codice@skull.codicefactory.com:9095	jesusmg	11/21/2018 16:26:00	devops: fix gluon BasicConfigurationTest false +
git /main/scm23591	codice@skull.codicefactory.com:9095	violeta	11/21/2018 12:43:57	guihelp: show help doesn't check already seen help

This was more than 18 thousand branches when I took the screenshot. Plastic doesn't have issues with that. It is designed for that.

Now, look at the following screenshot: this is the `codice` repo (the main repo containing the Plastic code) filtered by 2018 (almost the entire year).



What you actually see on the diagram (the upper part with all these lines) is the area marked in red in the *navigator* rendered at the bottom. Yes, all these lines just belong to a tiny fraction of what happened over the year.

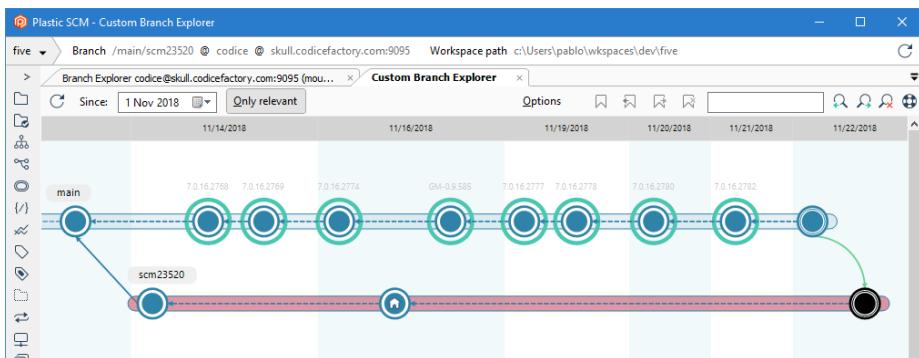
So, no, Plastic doesn't break with tons of branches, nor the tools we provide to browse it.

Ok, Plastic can deal with many branches, but how are we supposed to navigate that mess?

The zoom-out example produces this response very often. Yes, it can deal with it, but this thing you are showing me is a nightmare, correct?

Well, the answer is simple: except if you want to scare your colleagues, you'll never use that zoom level, same as you won't set the zoom of your satnav to show all the roads in Europe to find your way around Paris.

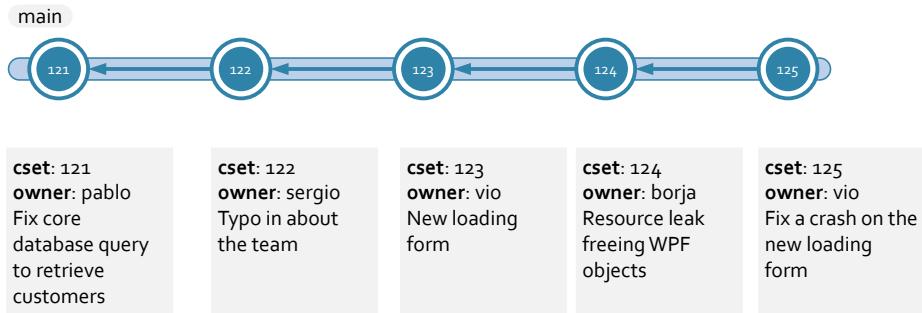
This is what I normally see in my Branch Explorer: just the branch I'm working on plus the related ones, **main** in this case. Pretty simple.



Task branches as units of change

We like to think in branches as the real units of change. You complete a change in a branch, doing as many checkins as you need, and creating a number of changesets. But the complete change is the branch itself.

Maybe you are used to think of changes in different terms. In fact, the following is very common:



This is a pattern we call *changeset per task* and it was extremely common in the old SVN days. Every single changeset is a unity of change with this approach. The problem, as you understand, is that this way you just use the version control as a mere delivery mechanism. You are done, you checkin, but you can't checkin for the reviewer, narrating a full story, explaining changes one by one, isolating dangerous or difficult changes in a specific changeset, and so on.

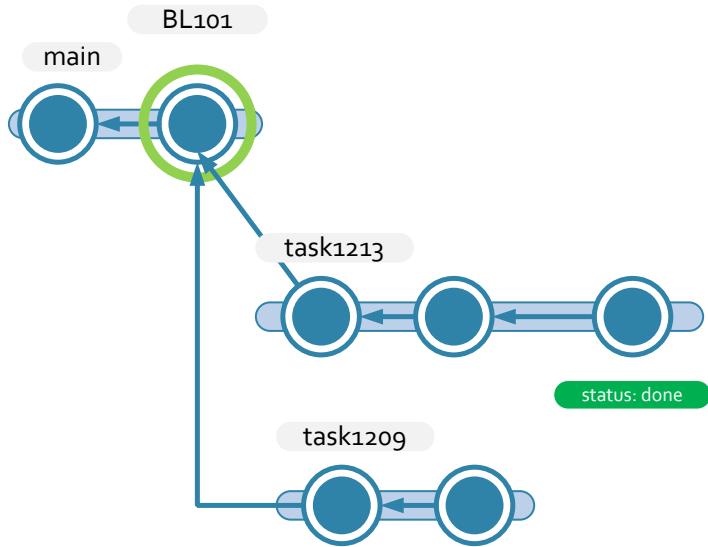
Instead, we see the branch as the unit of change. You can treat it as a single unity, or look inside it to discover its own evolution.

What happens when a task can't be merged automatically?

The purpose of task branches with full CI automation is to automatically test and merge (and deploy) tasks when they are done. The idea is that the integration branch, typically main, is protected so nobody except the CI system or the mergebot driving the process can touch it.

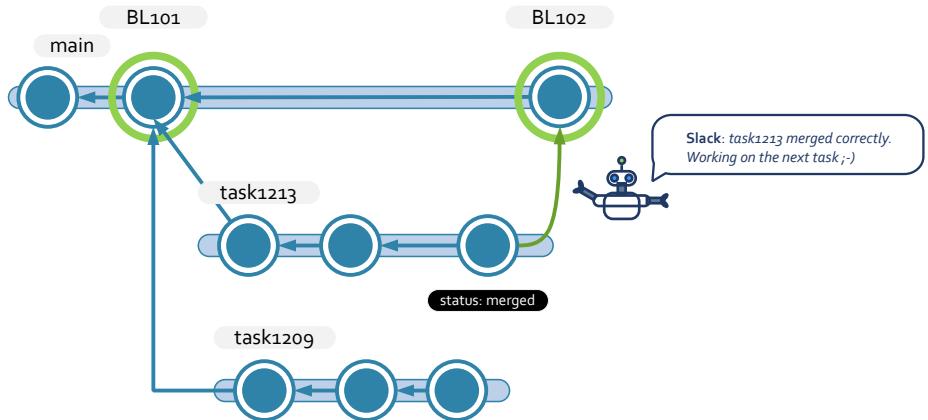
So, what happens when a merge conflict can't be solved automatically?

Consider the following scenario where a couple of developers —Manu and Sara— are working on two branches. Suppose at a certain point Manu marks his branch as done.

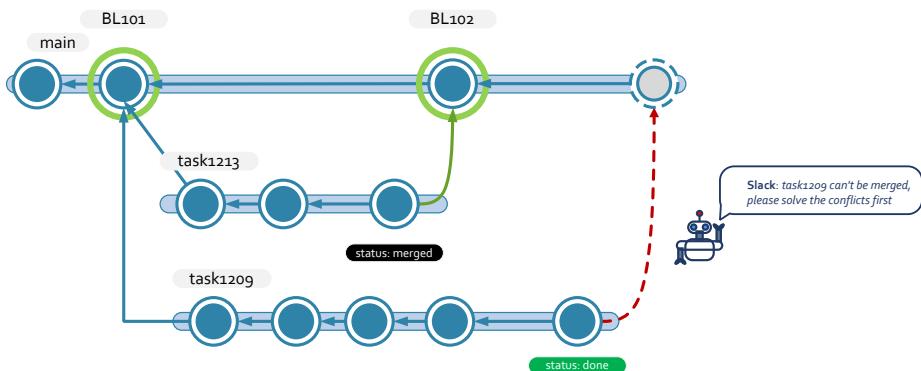


At this point the CI system monitoring the repo, or even better, the Plastic mergebot driving the process, will take [task1213](#), merge it to main in a temporary shelf, ask the CI to build and test the result, and if everything goes fine, confirm the merge.

The mergebot will notify the developer (and optionally the entire team) that a new branch has been merged and a new version labeled. Depending on your project, it is possible that the new version [BL102](#) is also deployed to production.



Now Sara continues working on [task1209](#) and finally marks it as finished, letting the mergebot know that it is ready to be merged and tested.

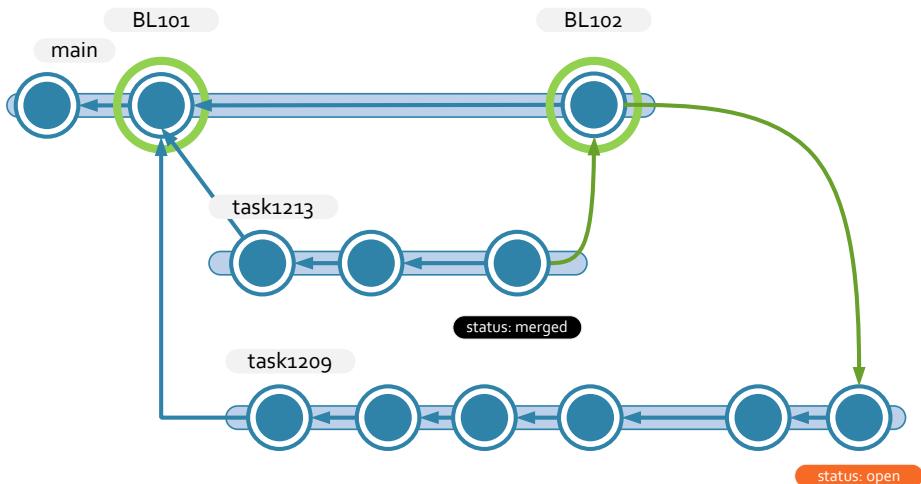


But unfortunately, the changes made by Manu, which were already merged to main to create **BL102**, collide with the ones made by Sara and the merge requires manual intervention.

The merge process is driven by the CI system or the mergebot, but none of them are able to solve manual conflicts.

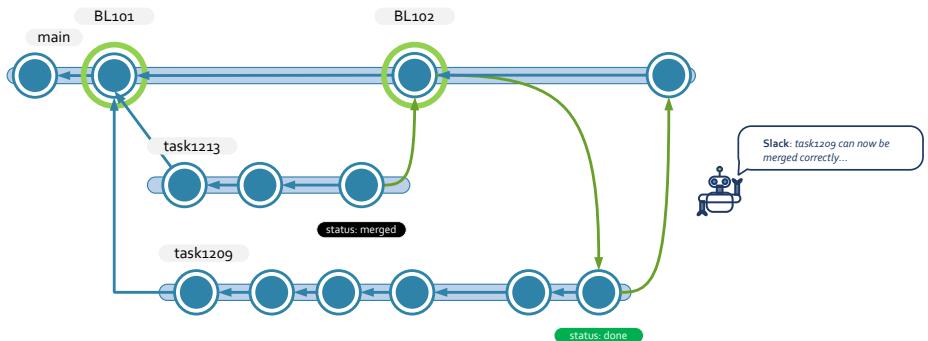
What is the way to proceed then?

Sara needs to run a merge from main to her branch to solve the conflicts and make her branch ready to be automatically merged by the mergebot or CI.



This is what we call a rebase in Plastic jargon, although it is quite different to a rebase in other systems. Check "["Plastic vs. Git rebase"](#)" in the merge section.

Once the conflict is solved, the merge can be automatically merged to main.



As you see, in a bot controlled integration branch, only branches that merge cleanly are introduced and used to build, test and deploy.

We don't delete task branches

At this point I'm sure is crystal clear for you how important task branches are for us. And, of course, we keep them around, we don't delete them.

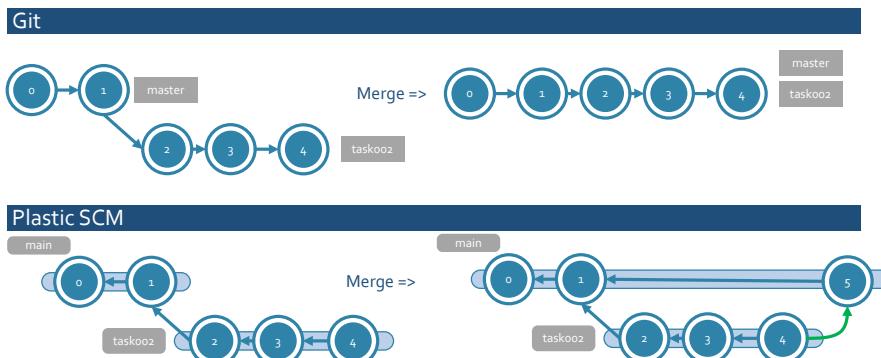
I cover this topic because users rushing away from Git frequently ask how they can delete their branches. They are as horrified at first when they learn they can't as we are thinking they want to delete their precious branches 😊

What follows is an explanation of one of the key differences between Git and Plastic, so if you are not really interested on some internals, you can definitely skip this section. There is also an even more detailed explanation [here](https://www.plasticscm.com/download/help/dontdeletebranches) [https://www.plasticscm.com/download/help/dontdeletebranches].

As I mentioned, in the Git world it is extremely common to remove branches once they are merged. There are several reasons for this.

1. Branches in Git and Plastic are different.

In Plastic, branches are changeset containers, while in Git they are just pointers.



This means that in Git is very easy to lose where a commit came from, while in Plastic, changesets always belong to a single branch and their history is preserved.

In the image above, after merge, you can't figure out if commit 3 was created in **task002** or master in

Git, but in Plastic commit 3 will **always** belong to **task002**.

We imagine Plastic as a wise librarian — preserving history for the sake of not repeating mistakes and creating lots of knowledge from previous work.

I really like to show the blame of a file, spot a given line, and immediately see which was the task branch in which it was created. This gives me lots of context. In Git you simply lose that.

Yes, that's why they like to squash branches, delete them and so on, but we have a radically different vision about all that. We like to tell a story on each checkin, to help reviewers speed up their work, instead of grouping tons of changes together into a single changeset ready to be merged. It is an underlying different viewpoint.

2. Most Git GUIs crash with lots of branches.

We have +18k branches in our repos. Plastic doesn't break. We have customers with way more than that. In Git, you need to delete branches because GUIs are not ready to deal with them. Open two of the most popular Git GUIs out there with as few as 1k branches and watch them crash. Not that they are mistaken or anything — they simply follow a different way of working. We optimized all our tools to be ready to deal with limitless numbers of branches.

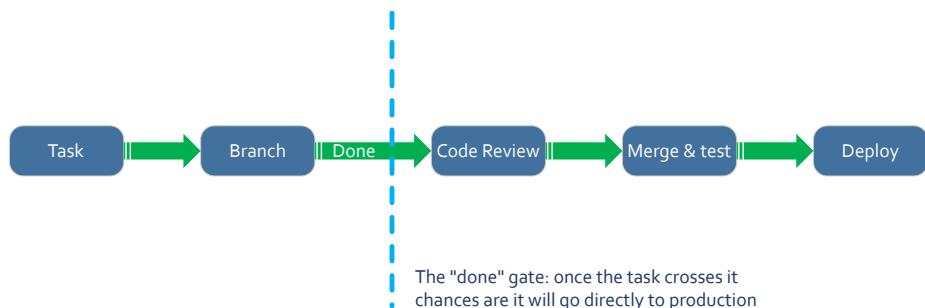
A finished task must be ready to be deployed

Here comes another thing to keep in mind: every task branch must be ready to integrate once it is finished. It might sound obvious, but it is a source of confusion in many teams.

"Yes, it is ready, but it can't be merged because..."

That's not good.

As a developer, once you set the task as "done", it means it is ready to be delivered. If the change was not good enough, if it is fragile, or if it will make the product behave in an awkward way, then the task shouldn't be set as finished.



You are crossing a point of no return (actually there will be a return if the task doesn't make the cut of code review, build, testing) and you must be ready for it.

It is a small price to pay for automation, but it is really worth it. The only thing is that the team has to get used to it. I can't stress that enough. Most of us grew up exposed to environments with lots of manual intervention. So, marking as *done* was not that crucial. It was simply "yes, I'm done, just a small detail to

polish before we deploy it and...". That's not enough anymore.

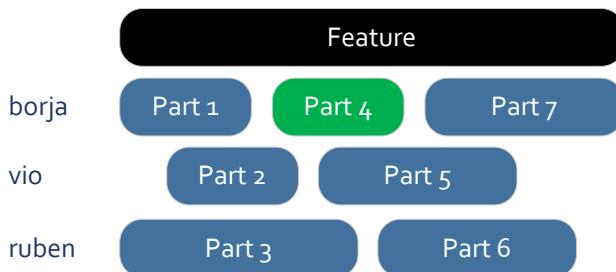
If you or your team are not yet there, try to get used to it quickly. Done means ready for production. It can feel overwhelming at first, but it is not, it is a matter of getting used to, just about being a little bit more careful. As return you get quite a big peace of mind because you know that moving your task to production is easy and fully automated, and not something that will require someone's attention (maybe yours) at 2 a.m.

Feature toggles

If you are already familiar with feature toggles, maybe the entire section was irrelevant for you ☺

The question we all ask when confronted with this "every task must be ready to deploy" is: what about parts of a bigger functionality that is not yet complete?

Because we already asked for tasks to be independent, which normally requires them to be carefully split. But now we came up with a new requirement: small **and** deployable, which sounds counter-intuitive.



You have a big feature split into 7 parts that will in turn be converted to tasks and implemented using task branches. How is it possible to deploy Part 4 if everything else is not ready?

Very easy: you merge it to main, and even deploy it, although it will be hidden. It doesn't need to be hidden for everyone; maybe you can simply enable the feature for the dev team, or for a small number of early adopters. At the very initial phases, if nothing will really work, it can be simply hidden for everyone.

What is the advantage then if nobody will see it? Why should it be deployed at all?

Hidden doesn't mean the new code is not passing tests on every release. By the time the full feature is ready to be activated, the individual parts have gone through testing several times. And exercises in individual exploratory tests. And the integration of the last piece won't trigger a big-bang merge, just a smaller part going through. And the full deploy cycle, whatever your actual system might be, will be fully exercised.

It is all about setting up a workflow and sticking to it all the time, not only when things are easy or hard. You stick to a way of working and exercise it that often that those long release nights simply vanish away.

Review each task

How we started reviewing every single task

Eons ago we tried to do formal reviews. You know, the entire thing with moderator, a short meeting, a scribe, and the author explaining. We only did it like twice. Too heavy. It wasn't natural, we didn't really adopt it, and it ended up always being postponed.

So, we changed our minds and started the "no task is merged if somebody else doesn't review it" around late 2012.

We suffered an initial slowdown for a couple of weeks, but then we all got used to it.

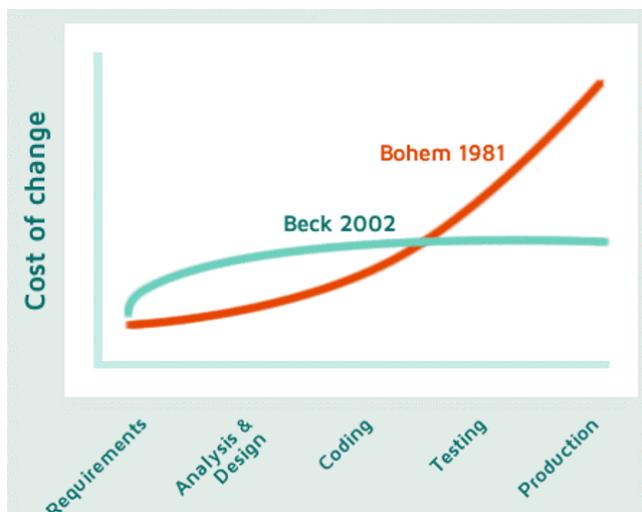
Code reviews proved to be great to train newcomers on "how things get done here", to keep some coherence in the design and to keep the code simple.

Reviews are crucial to prevent code from rotting

Now, fast forward to the present and the DevOps era. Remember the slogan: a continuous flow of **high-quality** changes ready to be deployed.

You only achieve real speed if the code stays ready to change. The key word is *stays*. Code rots. It gets worse with time after a few changes here and there.

I always like to share this graphic coming from two classics, Boehm's "Software Economics" and Beck's "Test Driven Development":



This is a nice graphic to help management understand why refactors and code reviews pay off 😊

Refactoring is the key practice, and code reviews only add to this effort. With every single finished task, there is a chance to make sure the actual design of the solution is fine, that the code is simple enough, and the style matches the rest of the team, etc.

Reviews to find bugs

Reviews are also great if they can stop new bugs before they reach the release. But, in my experience, it only happens if the review is very short. Really difficult on long tasks with many modified files, or many new ones, are extremely time consuming and hard to review.

And here it is where "[checkin for the reviewer](#)" kicks in: everything is much simpler with a little help of the developer.

Some useful tips:

- Separate the actual bug fixes from refactors (covered already).
- Separate *simple housekeeping* changes from the key ones in different changesets. This will also help reviewers focus.
- Mark changes you are unsure about. Don't be shy to drive the reviewer. Is there something you are not really sure about? Tell it. It can be a checkin comment, or a note on the task (your Jira?) saying "please, check cset 4893b715 because I'm unsure it will overflow". Otherwise the risk is that an important change will go unnoticed and the reviewer spends time on something pointless instead of the really important stuff. When we develop something big, with lots of new code, we practice this intensively to focus reviewer attention on what really matters. Nothing prevents him to read everything, but at least we all know the key things are checked.

How many reviewers?

It really depends on the team. The key is to have *at least* one so every task is code reviewed.

More than one can be great, at the cost of a bigger impact on everyone (context switches, more extra work that piles up together with actual coding) and an increase in task cycle time (more people involved to move it forward).

What we do very often is to always have one reviewer, but in especially difficult tasks, ask more than one colleague to review specific parts they know better. It is not something that happens every day for us, though.

How to actually do the reviews

What tools? I hear you asking. Well, we provide a built-in code review system in Plastic, and at the time of this writing we just finished designing a major improvement. So, you can definitely use Plastic's code review, or go for a third-party like Crucible.

You can even perform totally manual reviews. In fact, I've done this several times. I open a Plastic branch and diff changeset by changeset, and simply take notes in the issue tracker, in a new enclosure, of what needs to be changed. It is not super-fancy or anything, but it works.

Validation – exploratory tests on each task

A short intro to Exploratory Testing

There is a great book on Exploratory Testing that we all love here: "[Explore It!](https://pragprog.com/book/ehxta/explore-it)" [<https://pragprog.com/book/ehxta/explore-it>] published by Pragmatic Programmers.

Manual tests done by humans shouldn't be repetitive tasks. If they are repetitive and follow a script, then they should be automated. Of course, there are time and budget restrictions preventing this and many teams still do tons of manual testing, but in an ideal world, those are all executed by test frameworks.

What are Exploratory Tests then? Tests performed by humans, but letting humans do what they do best: imagine and improvise.

Simply put: Exploratory Testing is traditional structured random testing. Let me explain: it gives some rules so you can organize a manual testing process which basically consists on letting the testers arbitrarily test the software instead of constraining them with a script (which could be automated). For me, exploratory testing is the same for testing as agile methods are to coding; it can look at code and fix, but there's some order which prevents the chaos from happening.

Exploratory testing is based on well planned and time constrained sessions. The team plans in advance what needs to be tested and writes a set of **charters** or test goals. They define what they need to test but not how to test it (you can always give some ideas to help, but you can't create a script).

There are two important topics here:

- **Tests are planned:** which means you get rid of chaotic random testing.
- **Test sessions are time boxed:** you limit the time to test a given functionality or a part of your software, so you clearly state results should come at a fast pace. Of course, really difficult scenarios won't be so easy to constrain, but normally you can expect results to come during the first minutes/hour of testing, which will also help keep people concentrated on what they're doing.

Each tester takes a charter and starts testing the assigned part of software, creating a log called testing session in which they'll write how long it took to set up the test, how long they were actually testing, writing documentation and also exploring unrelated but valuable pieces of software (you start with something and continue with something else you suspect that can fail).

The testing session will record all the issues (minor problems) and bugs detected.

So, there's probably nothing special about exploratory (my explanation is very simplistic too) but it helps to organize a whole testing process; it tells you what you should plan and how (constrained times, normally no more than 2 hours per session), and also what results to expect and how to record them.

We record all the test sessions in our internal wiki, and link each issue/bug to our bug tracking system, setting a specific detection method so we can later on determine how many issues and bugs were detected using exploratory testing. Linking from the wiki to the issue tracking and back allows for better navigation.

Validation

We call it "validation", although it is too big of a name which can invoke the demons of formal validation or something, so I better not go there.

This is how it works: someone gets the feature or bug fix and ensures it does what it should.

It is important to highlight that "validations" are not manual tests. As I said, we are supposed to automate those.

The validation is just trying to figure out if what we did makes sense for the user, if it is usable, if it really fixes the bug (even if a unit test or GUI test was added).

It is some sort of short Exploratory Test that helps ensure we produce a usable and consistent product. A validation can be as short as 5 minutes (switch to the branch, build the code, run it, setup an example, etc.) or as long as 1-2 hours if the scenarios to test are very complex. But, the usual is just 10-20 minutes each at most.

By the way, sometimes we do longer (1-4h) exploratory testing sessions following what "Explore it!" describes. We generate the reports for each session, captured bugs, followed up with new tasks, etc. We do it when we have some new big features, but the actual daily practice is the short validations.

A small story on our experience with Exploratory

Back around 2012 we hired an external team to help us doing weekly Exploratory Tests, but it didn't work as expected. It was a big shock for us.

I mean, they were not finding the kind of issues we expected, and we struggled to find out why.

The problem was that the testing professionals were not used to most of the development tasks related to the daily use of Plastic SCM. We thought we were doing something wrong. But, at the end of the day, in my opinion, you need testers who resemble the kind of users your software will have.

And Plastic SCM users are... well, developers. They know how to use Eclipse, Visual Studio, the command line, doing branching, merging, diffing code, creating actual code... the sort of things the external testers we hired were not used to, and that's why we failed.

Nowadays, daily validations and code reviews are part of our day-to-day in development, and of course they are time consuming, but we considered that they are worth every minute. We try to organize our days to avoid interruptions, so many do the pending reviews and validations early in the morning, then the rest of the day can be spent on development.

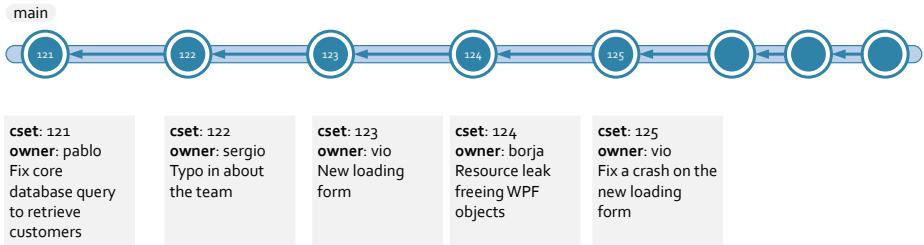
Some extra pros of task branches

I've described in detail the branch-per-task pattern and I've also talked about the task cycle and its main elements, so hopefully, you've already concluded why the branch-per-task approach is a good practice.

My intention now is to highlight, in a detailed way, why the branch-per-task pattern is the best way to develop and collaborate for nearly every team, nearly all the time (there will be circumstances where you won't need to branch that often, but believe me, it won't be so common).

Colliding worlds: serial vs. parallel development

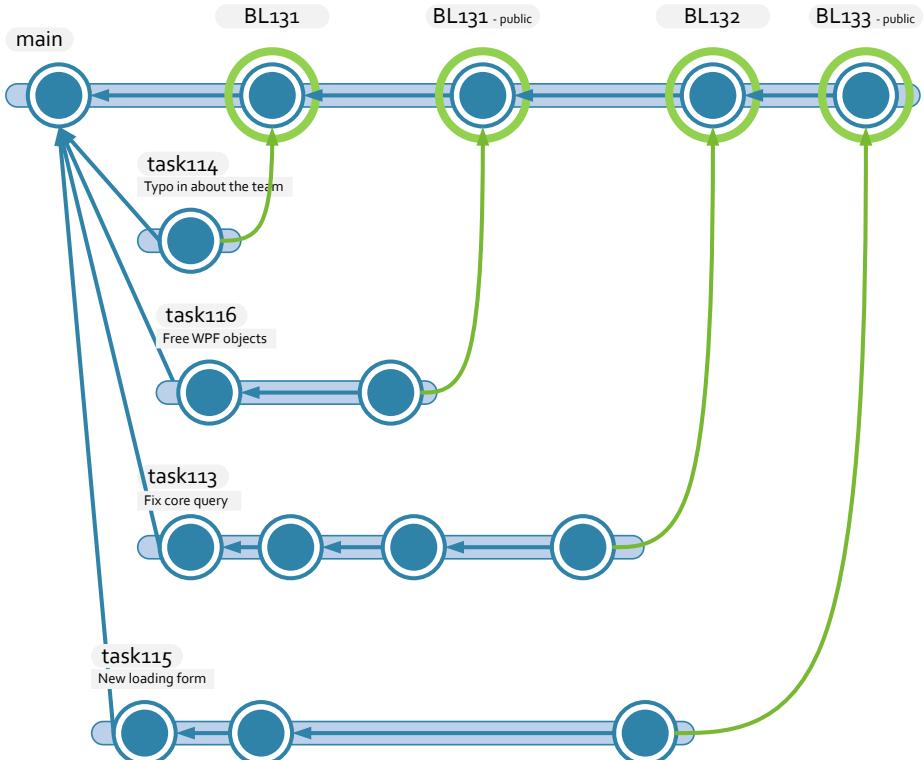
Let's take a look at a typical project following the serial development pattern, better known as mainline development (not to be confused with trunk-development). It just means there's a single branch where everyone checks in his or her changes. It's very easy to set up and very easy to understand. It's the way most developers are used to working with tools like Subversion, CVS, SourceSafe, etc.



In the example figure, we see how Vio creates a new loading form (`cset: 123`), but makes a mistake and then she has to fix it in a later check-in (`cset:125`). It means the build broke between `122` and `125`. Every developer updating his or her workspace in between would have been hit by the bug, and it most likely happened to Borja after he checked in `cset:124` and updated his workspace accordingly.

Also, if you look carefully, you'll see we're mixing together important changes like the one made on `cset:121` (big change on one of the core queries, which could potentially break the application in weird ways) with safer ones like the typo fixed in `cset:122`. What does that mean? It means that if we had to release right now, the baseline would not be ready or stable enough.

Let's see how the same scenario looks like using parallel development with the branch-per-task method:



As the picture shows, there are several branches involved since every task is now a branch, and there are merge arrows (the green lines) and baselines (circles). We could have created baselines before, but by using a branching pattern, you'll find it's much easier to know when to create them.

Using this example as a basis, I'll start going through the problems we can find in serial development, how to fix them with task branches, and why this parallelism is better.

Code is always under control

How often do you check in when you're working on mainline development? I bet you're very careful before checkin because you don't want your co-workers coming to your desk, shouting about the code not building anymore, right? Do you agree?

So, when you're working on a difficult task —something hard to change that will take a few days to complete— where's your code? Is it under source control? Most likely it won't be since you're reluctant to check things in that don't compile, that are not complete, or that simply collide with other changes.

This is a big issue and something pretty normal with mainline development; changes are outside of version control for long periods of time, until developers are completely done. **The version control tool is used just as a delivery mechanism instead of a fully-fledged VCS.**

With the branch-per-task approach, you won't have this problem: you can check in as often as you want to, in fact, it's encouraged. This enables you to even tell a story with your checkins as we covered in the "Checkin for the reviewer" section, to preserve your own development process.

But it was working 5 min ago!

I bet you've said that before! You're working on a change, your code is working, then you change something, it doesn't work all of a sudden, and you lose time trying to figure out what you did wrong (normally commenting and uncommenting code here and there, too). It's pretty common when you're experimenting with changes, learning an API, or carrying out some difficult tasks. If you have your own branch, why don't you check in after each change? Then you don't have to rely again on commenting code in and out for the test.

Keep the main branch pristine

Breaking the build is something very common when using mainline development. You check in some code that you didn't test properly, and you end up breaking some tests or even worse, introducing code that doesn't compile anymore.

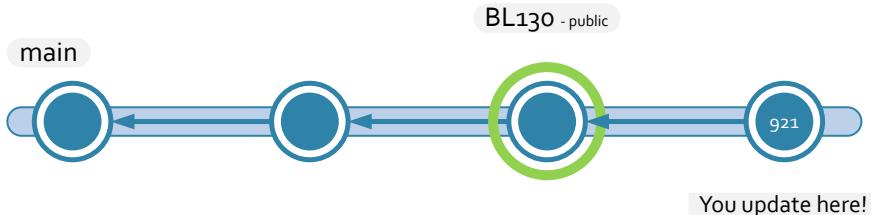
Keeping the main branch pristine is one of the goals of the branch-per-task method. You carefully control everything that enters the main branch, so there's no easy way to accidentally break the build.

Also, keep in mind that the usage of the main branch is totally different with a branch-per-task pattern. Instead of being the single rendezvous point for the entire team, where everyone gets synchronized continuously and instability can happen, the main branch is now a stable point, with well-known baselines.

Have well-known starting points - do not shoot moving targets!

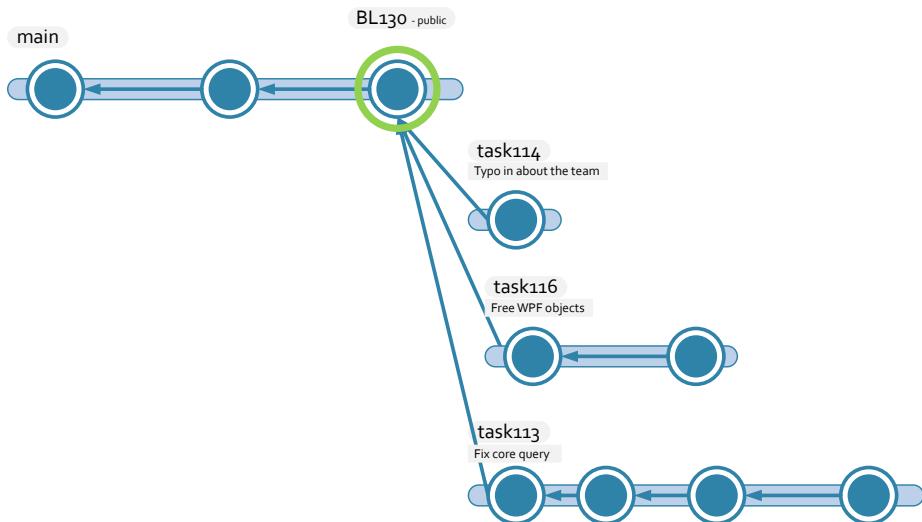
When you're working in mainline mode, it's often not easy to describe the exact starting point of your working copy.

Let me elaborate. You update your workspace to main at a certain point in time, as you can see in the following picture. What's that point? It's not **BL131**, because there are a few changes after that. So, if you find an error, is it because of the previous changes or due to the ones you just introduced?



You can easily say, "Well, if you're using continuous integration, you'll try to ensure the build is always ok, so whatever you download will be ok." First off, that's a pretty reactive technique - where you first break the build and later fix it. Secondly, yes, you're right, but still, what is this configuration? If you update at the indicated point, you'll be working with an intermediate configuration, something that's not really well-known - you'll be shooting a moving target!

Now take a look at the situation using task branches:



As you can see, there's a big difference. All your tasks start from a well-known point. There's no more guessing, no more unstable configurations. Every task (branch) has a clear starting point, a well-known stable one. It really helps you stay focused on your own changes.

Update with trunk-based devel.: I took this entire section from some writing I created years ago, dating almost from the very first days of Plastic. Most of what it says is still true, so it made sense to me to add it for clarity. Now, there is one major shift here: we used to do *manual integration* long ago, where every single task branch was manually merged to main by an *integrator* (or build master). Now we

automate all that. How does it change the *shooting a moving target* problem? Well, now every new single changeset in main is totally trustable because it passed all tests, there are no *intermediate integration changesets* anymore, so you are fine creating a new branch from each new changeset, no problem. Of course, the full *moving target* stays true for teams still stuck to *checkin to main* and done, because if that's the case, they might still be on shifting sands.

The psychological side: As mentioned above, it is super important to make sure every single task starts from a stable point. The reason goes far beyond coordination and preventing bug spreading. It has an impact on productivity. If a test fails in your new task branch after a few checkins, you know it is your stuff. It is not finding how's to blame. It is about the productivity impact. When something fails and you don't know for certain if it worked before, you'll have to spend minutes fighting your brain that already thinks that the bug was provoked by someone else. Funny but true. Didn't happen to you? But, if you are certain all tests passed in the base of your branch, you'll jump straight to fixing the problem you just introduced.

Enforce baseline creation

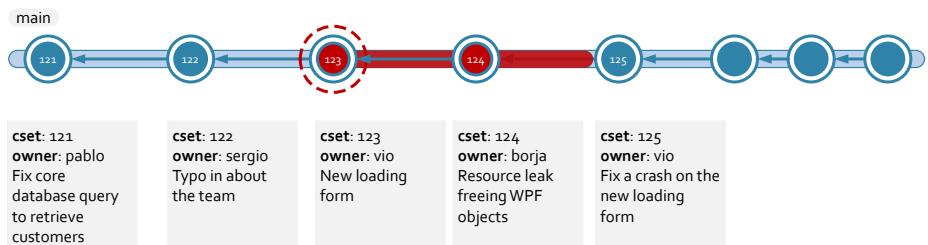
Creating baselines is a best practice. Using the branch-per-task method, baselines become a central part of your daily work. There's no better way to enforce a best practice than making it an integral part of your workflow.

Automation note: Yes, if you are already using full automation, this point will be more than obvious to you 😊

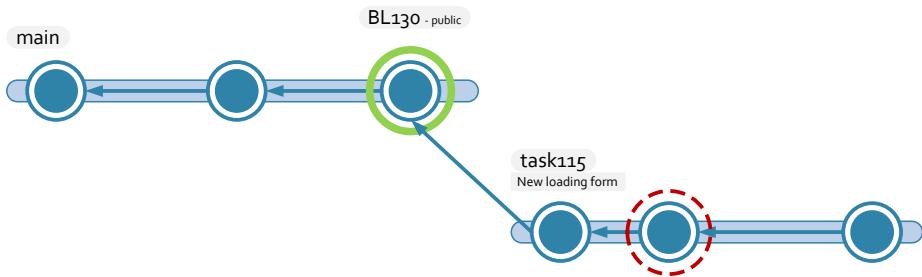
Stop bug spreading

People dealing with dangerous materials work in controlled environments, and usually behind closed doors, to prevent catastrophes from spreading if they ever happen. We can learn a valuable lesson from them.

Look at the following mainline example: Vio introduces a bug, and immediately everyone hitting the main branch will be affected by it. There's no contention, there's no prevention, and the actions are entirely reactive. You break the code and yes, you fix it, but the build should not have been so easily broken in the first place.



Now, let's take a look at the same situation with task branches. The bug will still be there, but we have a chance to fix it before it ever hits the mainline. There's contention and there's a preventive strategy in place.



Automated tests passing on each task branch

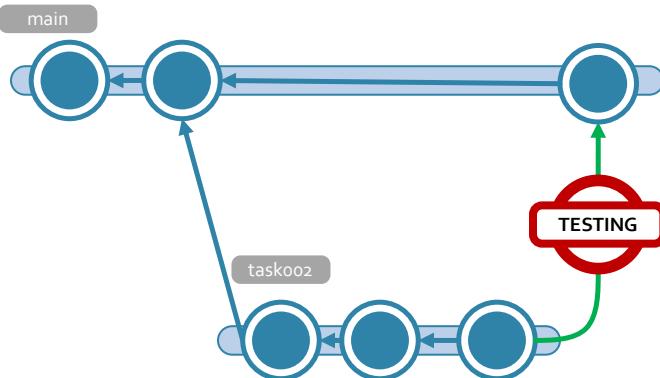
I can't stress how important automated tests are for task branches, and in fact for any software project.

Thanks to automated testing, we can refactor and clean up code, something key for a code base especially if you expect it to survive a few years.

If it wasn't for automated tests, it wouldn't make much sense to do task branches or even trunk-based development. What would be the *gate* to decide when a branch must be integrated? Just the code review? It wouldn't be enough.

Automated tests are the gate to main

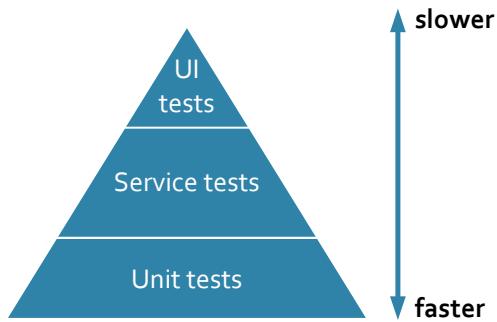
Every task needs to be reviewed (and validated optionally) prior to be merged. But the last step, the actual gatekeeper, is the automated test suite.



That's why you need a solid test suite to implement useful task branches, like it is for trunk-based development and DevOps overall.

The Test Pyramid

Check [this article](https://martinfowler.com/articles/practical-test-pyramid.html) [<https://martinfowler.com/articles/practical-test-pyramid.html>]. It is a concise explanation of the famous Test Pyramid:



Simple rule of thumb: write as many *unit tests* as you can, and as few *UI tests* as you can.

Want to learn why? I'm going to share our own experience on the matter.

Unit tests

Fast, reliable (unless you really write some messy ones) and capable of testing everything. Yes, if you think you really need an integration test because "it can't be tested with unit tests", I bet you are wrong. I was.

Now, we create more unit tests than ever and less and less GUI and smokes. Some are always needed, but the fewer, the better.

Service/Integration tests

We call them *smoke* internally, but they are sort of integration tests.

We extended NUnit long ago and created PNUnit (parallel NUnit) and we use it to sync the launch of client and server even on different machines.

Tests are written in C# and automate the command line.

Our mistake: Each test takes between 4-15 seconds to run. Too slow. We didn't realize at the beginning and created tons of them. Many are still in service.

Test slowness end up creating the "new versions are an event" problem, which is one of the key ones to solve when implementing true DevOps. Otherwise, new versions or releases are something painful that the entire team will try to postpone. And it all ends up in long nights and weekends. Not the way to go.

Fortunately, smokes are stable and false positives are not common.

UI tests

We also have a bunch of them. They can detect UI issues and help avoid regressions.

In our case, they need to start the server, wait for it to be ready, and so on. And well, **they are the root of all evil**.

I mean, unless you write them really, really well, they end up being fragile (the worst that can happen to

an automated test suite).

We rewrote many GUI tests over the years. This is how the whole thing evolved:

- First, we started with the "record and rerun" technique (offered by many vendors) which proved to be a nightmare (auto-generated code ends up being a mess).
- Later, we moved to just writing code to create tests, because, while more difficult to write at first (not really, but you do need programmers for that, and well, we are programmers anyway, so not a big deal for us), it ends up paying off because you have more maintainable tests that you can easily refactor and adapt when the UI changes.

But, even with coded GUI tests, we were hit by long tests, and by the unavoidable issue of "control not found" that comes with all UI Automation packages I know.

That's why we ended up writing our own small framework for GUI tests, but that's an [entire story](https://www.plasticscm.com/download/help/howtowork) [<https://www.plasticscm.com/download/help/howtowork>] unto itself.

Start small

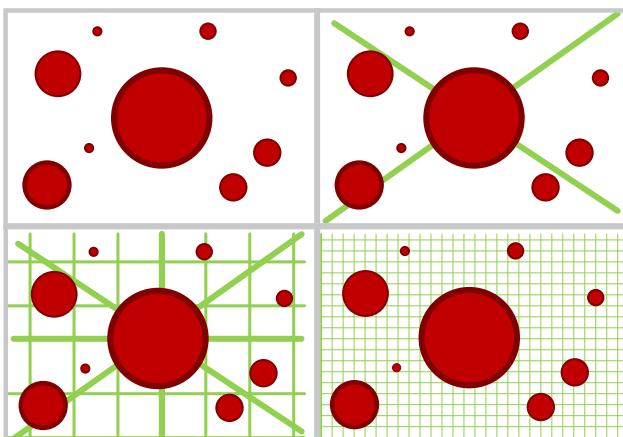
If you are reading this and your team already has a solid test suite with great test coverage, then you are all set.

If not, let me share a trick: start small. I've seen so many teams being discouraged by their total lack of tests. "We'll never get there" they thought. Of course, you can't pass from zero-tests to 10 thousand unit-tests by stopping all development and just focusing on testing for months. That's not realistic. Business must go on. So, what you can do? Battle inertia. The easiest is to simply continue with zero-tests. Write one today, have a few in a week. Keep adding. In a month you have a small number. In a year the code will be quite well tested. The alternative is to sit exactly where you are now in a year, which I bet is not such a good choice.

Automated tests are a safety net

A like to think of tests as a safety net that gets thicker and thicker as you keep adding tests.

The following picture shows what I'm trying to explain:



1. First you have no tests, and all kind of bugs go through.
2. Then you build your initial safety net. It is too wide so it only catches really big issues. That's good though, at least you are sure the really big issues won't happen again.
3. You keep adding tests to catch even smaller bugs.
4. At a certain point your test-net is so thick only really small bugs can escape ☺

The "safety net" example also gives you a good idea for the strategy to follow: focus on catching big issues first.

A couple of rules we follow:

- Every bugfix adds a test so this bug doesn't happen again.
- Every new feature adds new tests. Reviewers in fact reopen tasks if the new code is not tested.

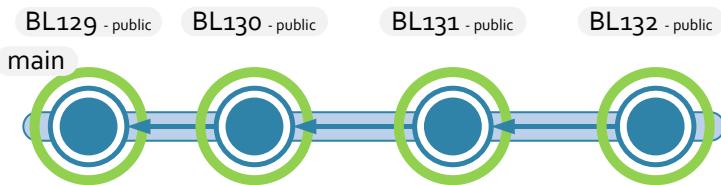
Both techniques help even if you don't have a test suite in place; they'll help moving forward.

Every release is a release candidate

In "[A finished task must be ready to be deployed](#)" we saw how, as developers, we must be ready to say goodbye to the task and let it fly on its own to release, without us doing any further work on it as soon as we mark the task as *done*.

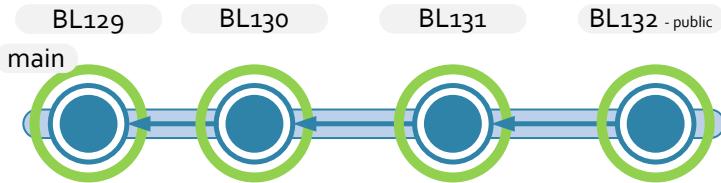
Well, this is because every single task merged to main must be ready to be released. The release machine wouldn't be ready for the DevOps league if new changesets on main are not really stable, really prepared, or "wait, yes, but we need to make more checks because...".

This means that a situation as follows would be perfectly normal:



For SaaS products, this would be desirable because your team will simply update the cloud software and users will immediately see improvements without any required upgrades.

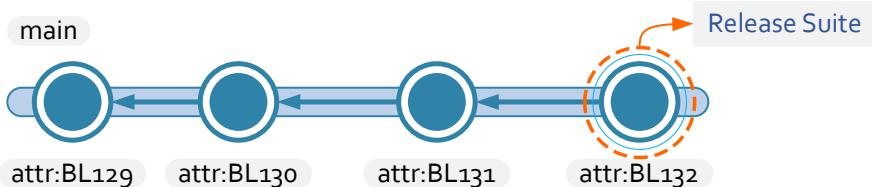
There will be cases, though, where publishing every release won't be that desirable because you don't want to run users crazy with 19 new releases every week. The solution is quite simple and obvious: don't make every release public.



Extra testing – grouping releases

Finally, there will be cases where you need to pass an extra test suite before publishing. It is not the ideal, but slow test suites, or the need to run load tests for hours before deploying the new version might force you to do that.

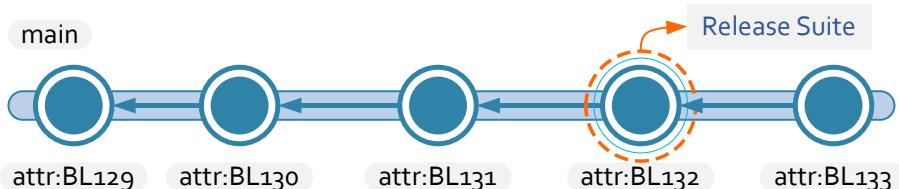
If that's the case, the situation will look as follows:



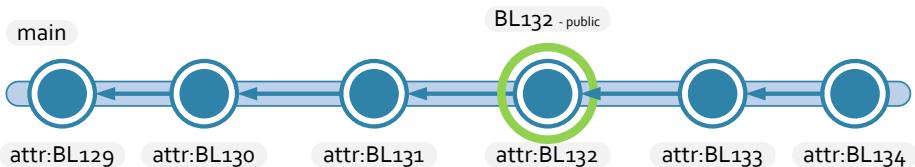
Teams can perfectly label each new build, but sometimes is better to save labels for real *versions ready to deploy* and use attributes for build numbers. We do this ourselves when releasing Plastic. Every single new changeset in main has a build number, but only real stable releases have labels.

As you can see, build **BL132** is used to launch tests grouping the new tasks merged from **129** to **132**.

While the test suite passes, new tasks can reach main:



Then, after the release test suite passes, the new version is labeled:



Consider this a big warning: This practice will create broken builds. The task test suite won't be complete enough because it requires a second test suite to pass, so the second one can detect issues that prevent the new version to be released. If that happens, you are in broken build. You'll need a new task to fix it and it must reach main as quick as possible to revert the broken build status.

The situation, though, is not as bad as breaking the build with every task, because your main might not be good enough to go public for everyone, but it can be still be deployed to smaller groups, or even to the internal development team, which is good for dogfooding.

Be selfish with tests and clean code

We can all talk for hours about how great having near perfect test coverage and pristine code is. It is good for the project, for the business, it is great to react fast to requirement changes. We all know that.

Let me share a different point of view: be selfish.

- Keep your code clean just because every skeleton in the closet will come back and bite you. We experienced it many times over the years.
- Run tests just for your piece of mind. Do you want to be constantly interrupted with super urgent stuff that needs to be fixed **now?** No? We don't either. Tests keep you safe from interruptions because the big, urgent things shouldn't happen.

Trunk-based development

What exactly is trunk-based development and how does it blend with task branches?

What is trunk-based development

It is a technique to ensure that all changes reach trunk fast, and every change is correctly reviewed and tested. You can find out more about it in [Hammant's super site](#) [<https://trunkbaseddevelopment.com>]. Trunk is the main development line, "main" in Plastic jargon.

Trunk-based development is the foundation of "continuous delivery".

It requires continuous integration; changes get continuously merged into trunk as soon as they are known to be good (peer-reviewed and tested with an automated test suite).

Trunk-based development is not the same as mainline development. Remember the old days with everyone just doing checkin to the main branch? (SVN anyone?). No, it is not the same thing. Trunk is all about making sure the build is stable, ready to be deployed. Nothing to do with the old mess.

Task branches blend well with Trunk-based development

Before you cringe, **trunk-based development is compatible with short-lived task branches.**

I wouldn't even call these branches "feature branches" at all because it might be misleading. A feature can take long to implement, so the branch grows too old.

I rather prefer "task branches", which sometimes is used interchangeably, although the meaning is quite different.

Tasks are short, or at least they should be. How short? 4h short, 8h short, ideally not longer than 16hours short.

Features are longer than that, so we just split them up. Task branches are great for that purpose and they blend nicely with trunk-based development.

Why do I insist on using task branches instead of just doing checkin?

Well, suppose you are working distributed. You can checkin locally and then push, can't you? This ends up being a "local branch" after all, and you have a place to checkin as frequently as you need before

hitting "main". You can do this with Plastic when you work distributed, of course.

Now, why should it be different if you are working centralized? You can create branches, checkin often, then set the branch as ready to be merged to main.

In both cases (distributed and centralized), short task branches bring lots of benefits:

- The task branch is the actual "change container". You are not tied to a "single checkin"; you can checkin as often as you need. (I tell a story in my checkins, so reviewers can go changeset by changeset following my train of thought instead of hitting the full diff. It is great for big refactors).
- You can easily review the code before it gets merged. It is a branch after all. No need to create other artifacts (like temporary checkins, or send the diffs around to a code review system). Just review the branch.
- The branch can be merged in a single step. If something goes wrong (build, tests, whatever), the entire branch gets rejected.

How do task branches blend with distributed development?

I didn't mention distributed development so far. The reason is that all the practices described so far apply equally to distributed and centralized development.

In Plastic, it doesn't matter whether you create branches on your local repos or in the central one. You can benefit from task branches on both.

The only difference is that, obviously, you'll have to push your local branch to the central repo in case you are working distributed. You can push as many times as you need, no worries with that, if your CI system is capable of taking only task branches marked as done. All our native CI integrations can do it: Jenkins, Bamboo and TeamCity.

Automation, orchestration and mergebots

I believe I made my point clear by now: task branches are great if they are supported by good automation. You need good automated testing, and a CI system in place.

You can implement DevOps + trunk-based development + task branches using Plastic and Jenkins, [Plastic and TeamCity](#) [<https://www.plasticscm.com/download/help/devopsteamcity>] and [Plastic and Bamboo](#) [<https://www.plasticscm.com/download/help/devopsbamboo>], or even integrate through the GitServer interface.

We wanted to make things easier for teams implementing Plastic, though. That's why we added *mergebots*. A mergebot is a piece of software that monitors your repos and trigger builds when certain conditions are met. It merges the task branch first, creates a *temporary shelf* with the merge results (if the branch merges cleanly) and then asks the CI system to build and test the *shelf*. If tests pass, the mergebot will confirm the merge and label the result.

The mergebot can notify each step to the team using email, Slack, or even custom created *notification plugs*. It can also consider task statuses from issue trackers and make changes on those status based on the actual merge result and test phase.

The Plastic *webadmin* interface includes a DevOps section to configure and monitor mergebots.

Learn more about mergebots [here](https://www.plasticscm.com/download/help/hireamergebot) [<https://www.plasticscm.com/download/help/hireamergebot>].

What about GitFlow?

If you never heard of GitFlow you can skip this section entirely.

But, if you are wondering why we prefer the task branches strategy better than GitFlow, then keep reading.

[GitFlow](https://nvie.com/posts/a-successful-git-branching-model/) [<https://nvie.com/posts/a-successful-git-branching-model/>] was introduced back in 2010 and since then, adopted by teams all over the world. It is about creating different long living branches called *master* (it would be *main* in our case), *develop*, and then *release branches* to articulate work.

I don't like it because I think it is overdesign for most teams. And I don't mean it is good just for teams doing very *complex* things. You can develop the most complex software and stick to a much simpler branch strategy.

- You don't need a *develop* branch, simply merge to main. It is much simpler.
- You don't need release branches, simply merge to main and release/deploy from there. DevOps changed my view here.
- GitFlow popularized *feature branches* and feature branches are not the same as task branches. In fact, they can be a problem most of the times because they tend to live far too long before being merged back to main. In contrast, task branches are, by definition, super short lived. The former encourages mini big-bang integrations while the latter are a perfect fit for continuous delivery.

Yes, there will be cases where you really need a more complex structure than plain *trunk-based* plus task branches, but most of the time they will serve you quite well.

How to learn more

There are a few resources that are a must to master branching:

- "[Streamed Lines: Branching Patterns for Parallel Software Development](http://www.bradapp.com/acme/branching/branch-creation.html)" [<http://www.bradapp.com/acme/branching/branch-creation.html>], by Appleton and Berczuk, the writers of the incredibly good "[Software Configuration Management Patterns](http://scmpatterns.com)" [<http://scmpatterns.com>].
- "[Trunk Based Development](https://trunkbaseddevelopment.com)" [<https://trunkbaseddevelopment.com>], by Paul Hammant. Trunk-based is a great way to keep things simple, blends well with task branches and it is what most teams use to implement DevOps.

REPO LAYOUT STRATEGIES

How many repos should you create? When is it better to organize different reusable components into different repos? What is a monorepo? What are Xlinks? And what are submodules?

This chapter tries to answer all these questions and a few more, always sticking to a conceptual approach to help you gain a deeper understanding of the version control techniques and practices.

What is a repository?

A repository is the storage of versioned content.

Too theoretical?

Let's try again. It is the *database* that contains all the versions of your files, plus the branches, the merges, the changesets with their comments, labels, attributes, the moves and renames.

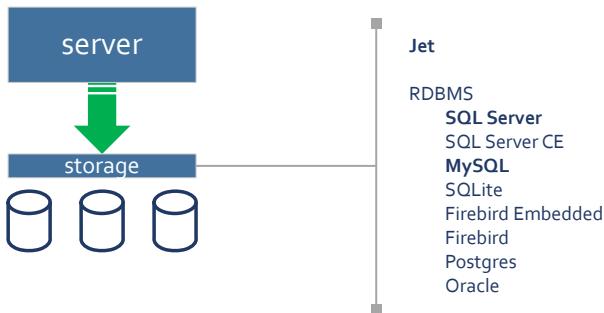
We make a key distinction between two conceptually different parts:

- Data: the actual bytes of each revision of each file. If you upload a *foo.c*, the actual content is the data. If you upload a 3GB .mp4 intro video for a game, those 3GB are the data.
- Metadata: file names, directory structure, comments, dates, authors, branches, merges between branches, labels... Simply put: anything that is not simply data.

Repository is a general concept used by almost all version control systems. Git calls it repository too, and for Perforce is formally *depot* but repository is a valid synonym too. You'll very often find it written as just *repo* for short.

Repository storage

How are Plastic SCM repos actually stored? It is not my intention to go under the hood and into the internals here, but just take a quick look into how repos *actually* store. Since chances are you have a strong technical background, I'm sure you'll appreciate understanding how things work. (If not, feel free to skip this, I don't want to bore you with gory details).



We have two main types of storage in Plastic SCM. They are completely transparent outside the server. I mean, when you are doing a checkin, or merging, or doing diffs, or even running a replica, you don't really care whether your server is storing repos in SQL Server or Jet. In fact, servers talking with each other don't care either. Client-server and server-server communication is totally independent from the storage backend.

That being said, the two major storage types are:

- **Jet.** It is our own ad-hoc, heavily optimized storage. It was first released in late 2016 after years of sticking to traditional database systems and trying to avoid reinventing the wheel at all costs. But, finally we admitted that having our own storage, specifically designed to deal with Plastic repos, would get much better performance results. **Today, Jet is the default backend for repo storage**, both on tiny single user installations and in our largest customers with terabytes in each repo and thousands of concurrent users.
- **Standard database systems.** We support a number of them although the server is only optimized to work with two for heavy load: MySQL and SQL Server. SQLite is good for single users working distributed and for really small teams. We don't recommend any of the others for big teams, not because they can't scale, but because we didn't optimize the Plastic server to work with them.

If you want to learn more about the story of Jet and why we finally reinvented the wheel, [read this blogpost](#) [<https://www.plasticscm.com/download/help/jetstory>].

Number of repos and maximum size

There are no limits on the number of repos and maximum size of each repo in Plastic SCM, other than what the physical disk limits imposes.

- You can have as many repos as you need.
- Each repo can be as big as you need. We have customers with repos as big as 5TB.

One project, one repo

Let's start simple. If you have one project, create one repo to host it.

There are teams with many different independent projects, something as follows:



We have a customer with more than 3000 developers on the payroll and their main server hosts more than 2000 repositories. They have many small teams working mostly independently from each other on different repositories. Most repos contain software that is deployed independently (sort of microservices) while it evolves over the years.

Another corporation in a different industry has more than 4000 repositories on the main server. Not all of them are live since many contain projects that were already finished and no longer evolving.

There is another one with more than 1000 developers, part in Europe, part in South America. They also have more than 4000 repos. Most of them contain *small* projects developed by teams ranging from 2 to 20 developers. Each project is an application that is deployed independently to production, controlling a different part of the business, or providing different interfaces to their stuff. They develop enterprise software for their own business. Some applications have requirements to be deployed together, but their repos stay independent and they control those dependencies from their CI systems.

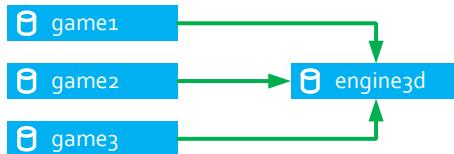
Not all our customers are so large. We have teams with less than 10 developers who work on projects for different customers and created several dozens of repos over the last 10 years.

What I described so far were independent repositories. It doesn't really matter how big or small they are. They do not have dependencies on each other. It means they are quite easy to understand and manage.

Xlinks: Reusable components

Things get more complicated when you need to deal with reusable components.

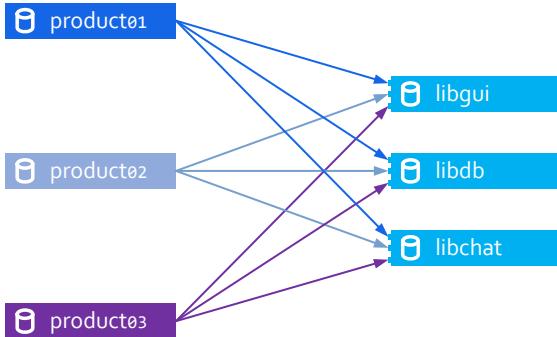
The following is a very common scenario for teams developing games, although the pattern is general for any reusable component.



All different games created by the company share a 3d engine evolved by the same or a separate team. It doesn't make sense to *copy* the same code in all games, of course, so the logical option is to separate it in its own repo and *xlink* to it.

An Xlink is a Plastic artifact to link repositories. Think of it as a special *symlink* between repos. I'll be talking more about it later, and you can read our guide about Xlinks [here](https://www.plasticscm.com/download/help/xlinks) [https://www.plasticscm.com/download/help/xlinks].

The challenge with repositories and components is that things can get complicated quickly as you can see below.



Here we see three products all sharing the same set of components. The number of Xlinks to create between them grows. From a conceptual perspective this can be a very perfect solution, as long as it doesn't become a nightmare to maintain.

Keep it simple - Don't overdesign your repo structure

Component repos and Xlinks are great tools, but as Uncle Ben said: *with great power comes great responsibility*.

The big risk is to start seeing components everywhere, start dividing the codebase into too many repos too soon, and end up in a big unmanageable mess.

This is something we have seen so many times over the years in all kind of teams. They get excited about isolating repositories and discovering components, and they create a bigger problem than the original tangled monolithic repo had.

So, yes, you can have many different repos, but use them with care.

Monorepos – don't divide and conquer

Monorepos have been gaining popularity over the past few years. [Legends say](https://trunkbaseddevelopment.com/monorepos) [https://trunkbaseddevelopment.com/monorepos] that Google hosts its entire gigantic codebase into a single mega large repository.

Why on earth would they do that?

- **Bend repository boundaries.** The boundaries of a repo are strict. It means if you put something on a repo, it is not easy to move it to a different one. Of course, you can always copy the files and checkin to a different repo, but loosing history and introducing the risk of duplication.
- **Shared knowledge.** If you split the overall *project* into pieces many teams and individuals will just be aware of the few parts they work on. They'll never go looking into the rest of the repositories for something they need, primarily because simple tools like searching and grepping won't help them. Very likely this will end up in duplication of code and efforts.
- **Easier to refactor.** Suppose you have a component used by 30 different projects. If you just download the repo that contains this component, it won't be safe to apply a rename to a public method or class used externally without a big risk of breaking one of those 30 projects potentially

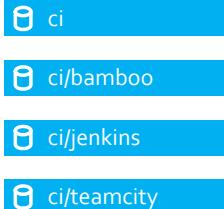
using it. If everything is on a single repo, you'll have the 30 projects on disk and the refactor will be easier.

According to Paul Hammant in his great [trunk-based development site](https://trunkbaseddevelopment.com) [<https://trunkbaseddevelopment.com>], Google, Facebook, Uber and Netflix all use monorepos. And, you know, when famous giants adopt a technique, it gains some credibility.

Submodules

There is one more thing to add to the repo tool belt: submodules.

Submodules are a way to organize repositories in hierarchies. In fact, the main practical advantage they introduce is defining a namespace in your otherwise flat repository naming structure.



As you can see in the figure, you can have a repository *ci* and then many children repositories *ci/bamboo*, *ci/jenkins*, with as many levels of nesting as you need.



Tip for teams using relational databases as storage

Although we are now mostly standardized on Jet, there was an advantage for large teams using relational databases as storage. Submodules are stored into the same physical database as the main repo they belong. This is interesting when you need many thousands of repos and your IT department complains about managing thousands of databases. Submodules are quite practical in that circumstance.

Practical advice: Fantastic repos and when to create them

After going through some of the basics, let's see if I can give you some practical advice that helps you decide how to structure your projects in Plastic SCM.

Keep it simple

This is a mantra in software development. Try to keep things as simple as possible. Ask yourself five times if you really need to create a component repo.

Overdesign will normally bite you harder than a too simplistic approach.

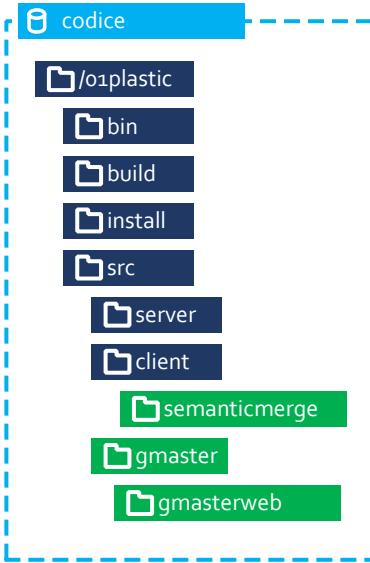
Monorepo are fine

I have to admit that before reading about monorepos as a legit approach, I always thought putting everything on the same repo was a little bit of a mess.

But, let me share how we structure our own main repo to illustrate how we implemented monorepos internally.

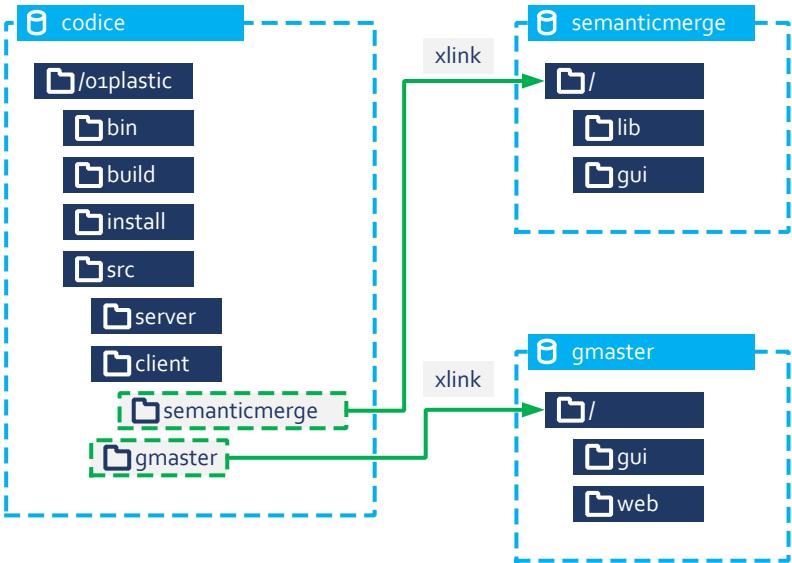
We work on one main product: Plastic SCM. But at the time of writing this we have other two secondary products: [SemanticMerge](http://www.semanticmerge.com/) [<http://www.semanticmerge.com/>] and [gmaster](https://gmaster.io/) [<https://gmaster.io/>].

We manage the three on a single repo called *codice* as you can see below:



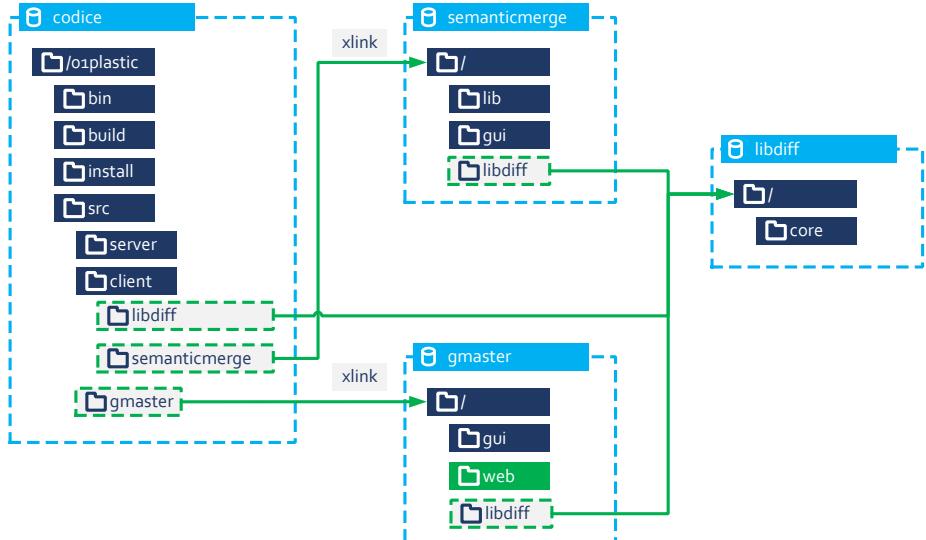
- SemanticMerge uses part of the code under *client* to take advantage of some of the text merge algorithms in Plastic. Also, some of the actual *semantic* core is under *client* too, outside the *semanticmerge* directory because it is used by both Plastic and Semantic.
- gmaster uses *semanticmerge* entirely, including common parts shared with Plastic.
- Plastic itself uses *semanticmerge*.

We could have certainly structured it differently with Xlinks as follows:



But since both gmaster and SemanticMerge require pieces inside 01plastic/src/client to build, then the two new repos wouldn't build in isolation. We can admit that, fine, they only build with mounted inside *codice*. But, if that's the case, what's the point in isolating them? Plastic doesn't really have repo size issues, so the division would be quite artificial and impractical.

Another alternative would be to split the repositories differently, so both semanticmerge and gmaster can be built separately from the main codice repo. A stricter separation in components would be as follows:



I marked *web* inside gmaster in green because this could also be an opportunity for even another Xlinked repo.

We can continue our splitting flurry but the question is: does it really make sense? Plastic can certainly cope with all those repos but, does it really make sense for the team?

In our case the answer was no.

Everything was simpler with a single code repo. Even the website for gmaster is on the same repo because this way it is very easy to build the same REST API definition both in the client and the server and refactor accordingly instead of splitting. It could be a good candidate for a different repo but, why?

I don't mean it is good to have a big mess in the repo. We are super strict with directory and file naming, and with what goes into each directory and so on, and we frequently reorganize directories when things don't look nice anymore. But a single repo serves us well.

Sometimes you need Xlinks

I explained my view about how important is to keep things simple, but of course there are cases where splitting and xlinking is really the best option.

How can we distinguish the scenarios?

It all depends on practical and organizational restrictions.

- **Size.** Can your organization afford having everyone on the same big repo or is it too large that it doesn't even fit on the developer machines? Sometimes, especially in game dev, we find repos so big that a single full working copy would be several terabytes. They could *cloak* the repos to only download parts of it, but splitting sometimes is easier to understand to avoid issues.
- **Historical reasons.** Some teams already had many different repos in their previous version control. They are used to that, their build files are ready for that, and they want to minimize the impact when switching to Plastic.
- **Privacy.** Several teams share the same 3d engine but each of them develops a different game and they are not allowed to see what others are doing. They all contribute to the engine with changes, and they can all benefit from improvements, but seeing each other's code is not doable. The same scenario applies to different companies contributing to a bigger overall project, where each of them are only allowed to work and see certain parts.

Conclusion

Plastic SCM repositories provide many options to model your project structure.

Our advice is: try to keep it as simple as possible. Mind the monorepos, they are widely accepted by the industry and they can be shockingly simpler than other more componentized alternatives.

CENTRALIZED & DISTRIBUTED

Should you work centralized or distributed? Which mode is better? Is it possible to combine both? What would be the advantages? What is the difference between an on-premises server and a Cloud solution? What are proxy servers?

These are some of the topics we'll be discussing in this chapter.

Centralized and distributed flavors and layouts

What is distributed and centralized

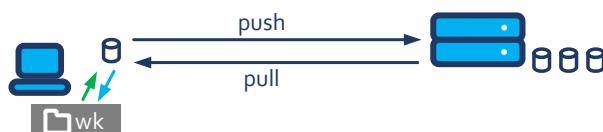
If you have previous experience with Git and Perforce or SVN, you know what distributed vs. centralized means. Otherwise, check the following explanation.

Centralized means you directly checkin to a repository (repo) that is located in a central server, typically on-premises.



Subversion, TFS, Perforce are typical examples of the centralized workflow. Plastic can work in this mode too.

Distributed means you always checkin to a local repo in your machine, then *push* your changes to a remote repo on a server.



Distributed is great because every checkin is local, so it is super-fast (no network latency potentially biting you). The downside is that you need two operations to reach the central repo: checkin and then push. This is not a big issue, of course, considering the benefits, but our experience tells us that for certain profiles it is like torture. In video games, for instance, artists *hate* push and pull. Checkin and done is more than enough for them.

Running a local server

Now, if you have a Git background you would say: "yes, but Git does distributed *without* a local server".

You are right, we need it because traditionally our server start time, while super-fast, was not fast enough to have zero impact if started and stopped with each command you run.

Nowadays, with Jet as the storage backend, we could probably go in that direction, but this is something we never really targeted because running a local server doesn't have a real negative impact.

In fact, there are other things this design allows that have been out of Git's scope for a while. In Git, a repo and a working copy are heavily tied together.

Until very recently, it wasn't possible to have several working copies connected to the same repo.

This has always been possible in Plastic. In fact, it is also possible to directly connect a workspace to a remote repo, something still out of the scope of Git and the source of many issues with large projects.

Plastic can do centralized and distributed

Remember, we named Plastic this way because we wanted to be flexible (yes, lots of plastics are very rigid 😊). So, Plastic can do both. It can be as fully distributed as Git is, or work fully centralized.

To do distributed, we need to provide *local repos* and that's why Plastic runs a super lightweight local server. It is the same codebase that can scale up to manage thousands of concurrent users and millions of requests per day, but tuned down to be as quiet and resource constrained as possible.

Is distributed better for branching and merging?

There is some confusion about this that I think is worth covering. Some developers wrongly believe that you need a distributed version control to do proper branching and merging.

That's why only a few version controls were good with branches before 2005. That was the year when Plastic SCM, Git and Mercurial were born. All of them are distributed, and when Git got worldwide popularity, everyone tended to think good branching was a DVCS thing.

I'm a version control geek, so I must say that while SVN, CVS, Visual Source Safe, Perforce and a few others were truly weak with branches and merges, Clearcase had a pretty high bar. It was very good, but it was hard to install, extremely expensive and after the IBM acquisition, got more bad press than good. But it was one of the strongest systems ever made.

Getting back to the point, you can do super strong branching and merging with Plastic while working totally centralized.

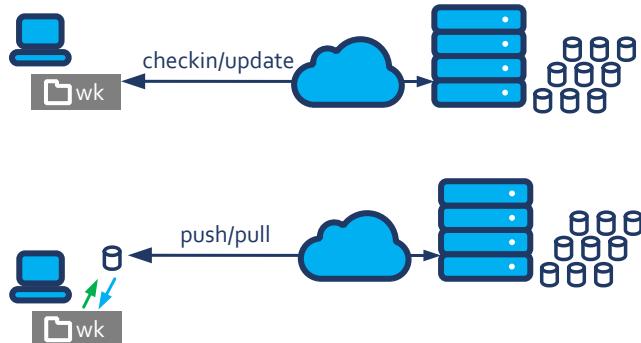
On-premises and Cloud

You can install your own Plastic SCM server at the office. This will be an on-premises server. This might sound obvious but I think it is worth making it crystal clear, especially for non-native English speakers 😊

Then, we offer a service to [host repositories in cloud](https://www.plasticscm.com/plasticscm-cloud-edition) [<https://www.plasticscm.com/plasticscm-cloud-edition>], managed by us, so your team is freed from the sysadmin burden. That's the Cloud option.

What impact is there when choosing Plastic Cloud vs. on-premises on the distributed/centralized strategy?

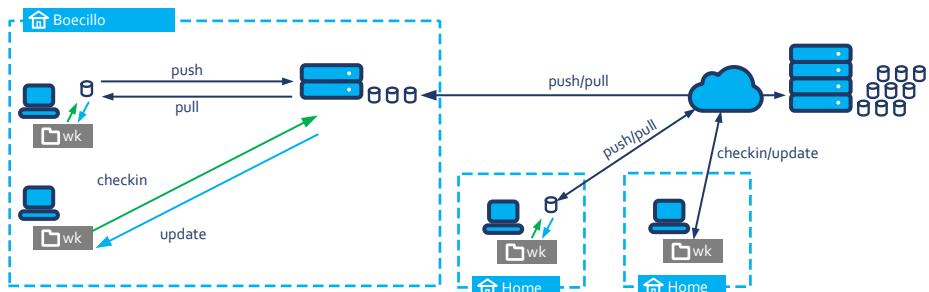
Actually none. You can work perfectly centralized or distributed with Cloud or on-premises. The only downside you might have is the added network latency if you compare Cloud with a server hosted on your office.



Mix distributed, centralized and Cloud

Something you must keep in mind when designing the actual version control layout for your company is that Plastic gives you a range of choices. You don't have to force the entire team to stick to a single pattern. Some can work centralized, some distributed, some connected to cloud, and you can even create multi-site setups.

The following diagram is an example of a possible setup:

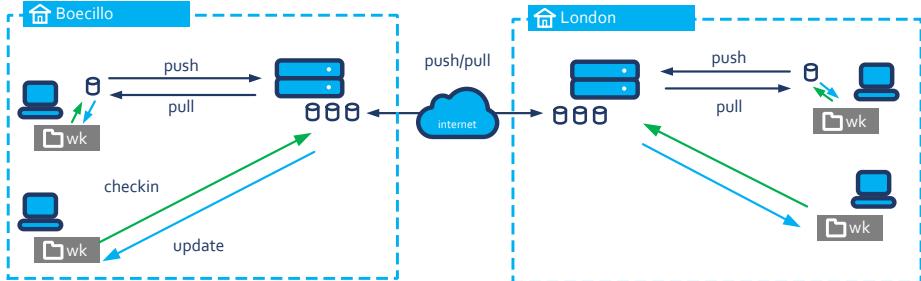


- There is a co-located team with an on-premises server where one team member is doing direct checkin while the other prefers to work distributed.
- The on-premises server push/pulls to Cloud. Cloud can be used as a backup of the central site or simply as a better way to give access to external contributors to the project.
- Then there are two team members working from home: one using distributed, the other sticking to direct checkin.

Multi-site

Multi-site stands for having multiple on-site servers connected to each other through replicas.
(Important: We call replica to push/pull, nothing new.)

The scenario can be as follows:



- Two teams.
- Each has their own on-premises server.
- Team members at both sites checkin locally or do distributed, but benefit from the speed of a close on-premises server.
- Servers push/pull between each other to keep fully or partially in sync.

Multi-site was an especially important concept before distributed version controls were widely adopted, as almost the only way to connect different sites.

Is multi-site still relevant nowadays?

- In reality, it is just a different form of distributed.
- Many teams can benefit from a super simple workflow sticking to direct checkin to a local server.
- And then different servers are kept in sync using replica (push/pull) triggered by a script that can run continuously, or just a few times a day, depending on what the team needs.
- Of course, if every single team member works distributed, there wouldn't be a big need for on-premises servers on each site, except if you need to run some CI tools locally.
- Cloud can be added to the previous picture as a third site that can act as single source of truth.
- Many other sites could be added to the picture.
- The on-premises servers are connected to each other through the internet. They can have an open SSL port, or bypass proxies using Plastic Tube, our own P2P protocol. Of course, any commercial VPN or tunnel software can be used to create the virtual link between servers.

Recommended layout

My recommendation is the same I made for branching patterns and repo layouts: try to keep things simple, both from an IT perspective and for the team members actually using Plastic on a daily basis.

What I mean by this is:

- If all your developers can use distributed, then there is no big reason not to go for it. Use a Cloud server as single source of truth and you are done.

- Do you really need an on-premises server? If not, stick to a Cloud one. If you really need it, then yes, go for it. It won't require a lot of maintenance, but always more than if you don't have one.
- You have multiple teams on multiple sites. Can they all stick to distributed? Then, maybe you are good with a single central server and they all push-pull to it. If not, then it is good to go for multiple sites and servers on each.

In short, Plastic gives you all the flexibility, but it doesn't mean you have to use everything ☺

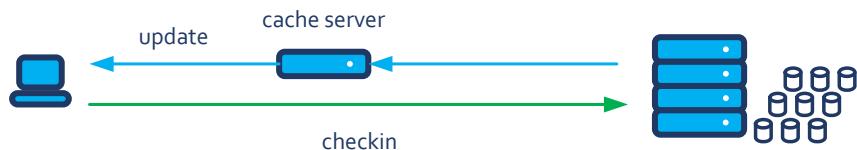
Proxy / cache server

You have another choice to connect distant sites: the cache server, a.k.a. proxy server.

It is a very simple piece of software; it caches data (not metadata), so it doesn't have to be downloaded from the distant server each time, saving network bandwidth and time.

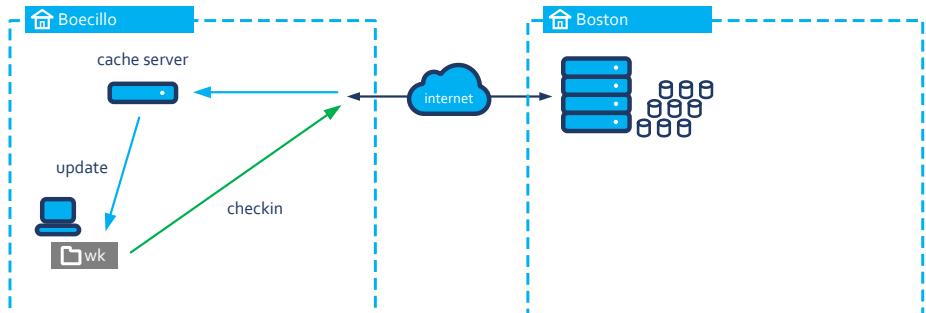
The following diagram explains how the Plastic clients (GUI or command line) and a proxy/cache server work:

- Checkin goes directly to the central server, so the proxy doesn't speed up the operation.
- Update tries to download the data from the cache server, if the data is not there, the cache server will download it from the central server and cache it for the next request.



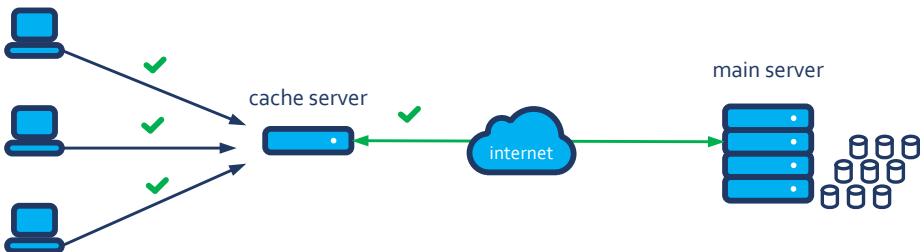
As you can see, the functionality is very simple.

The typical scenario is a co-located team that normally works with big files and is connected to a distant server.

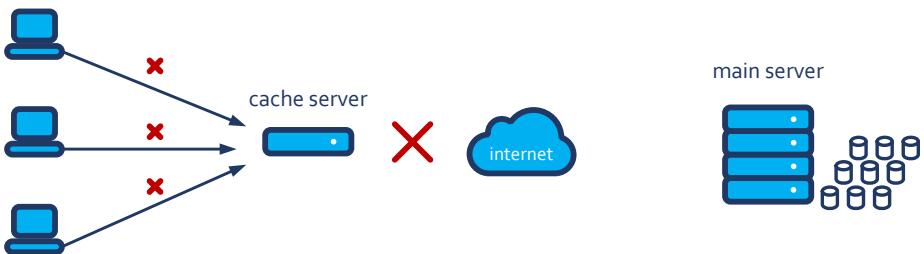


The team at Boecillo will benefit from faster updates (data downloads) if they have a proxy on-site because data won't be downloaded all the time from Boston.

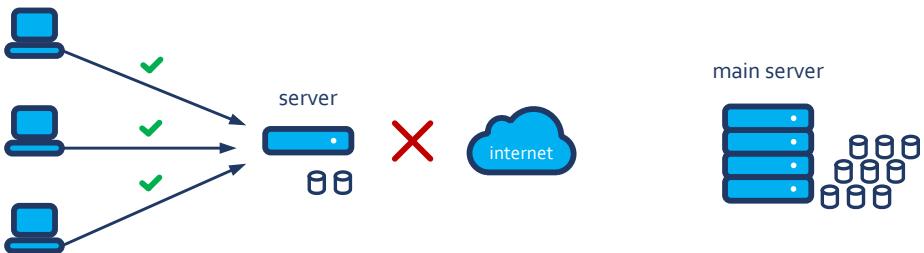
The scenario works well as long as the network connection works as expected. **Proxies are a performance improvement, not a solution for multi-site.** It is important to keep this in mind when you design your Plastic server infrastructure.



The downside with proxies is that if the network connection between the cache server and the central server goes down, then the users won't be able to operate normally. They won't be able to checkin anymore, and updates will also fail because only data is cached, not metadata.



Having your own repos on your machine, or your own on-site server for the entire team (we call fully distributed for the first option, multi-site for the second) allows you to have a solution resistant to network failures. The team can continue working even if the connection to the central server goes down.



If distributed and push/pull operation (a.k.a. replica) is superior to proxies, then why do we support them?

- A proxy is just a way to support centralized operations with some network optimization. You might have a distant team that refuses to do push/pull because of whatever restriction. This is very common with non-developers: they prefer to simply checkin, and dislike having to deal with an extra push/pull. A proxy might be a good solution if there is nobody on their site capable of managing a full server.
- If you need to enforce non-concurrent operation on certain files (locking) then you need to stick to centralized, and proxies can help distant teams.
- Network performance: we have seen heavy load scenarios where customers decided to install a Plastic cache server on different network sections to decrease the network usage of the main server. Nothing to do with distant sites, all happening inside the same building.

In short, we strongly recommend using distributed if possible, but proxies are always an option when replicas are not possible for any reason.

It is important to note that because some teams, especially the ones coming from Perforce, tend to prefer proxies over distributed because that is what their previous version control provided as a multi-site solution. Please, keep in mind the options and remember Plastic can do really distributed when needed.

The story of why we developed the cache server

Proxies are a well-known in the version control world. Perforce implements something similar (more functional than the Plastic proxies because they try to solve some things we do with distributed), and if I remember correctly even Clearcase had a solution along the same lines.

But, we developed the original cache server to improve end-to-end performance in a huge team of 1000+ developers in South Korea working on a single huge code base. The problem they had is that the traffic on the central server was so high that the network link was saturated. Most of the traffic was data being downloaded.

So, with their help, we decided to implement small *caches* on different network sections to reduce the traffic to the central server. And it worked.

The interesting thing is that all of this happened inside a single building! We were not targeting multi-site scenarios, we had replicas for that.

What is next for proxies

There is an idea we have been tinkering with: have a small proxy/cache on each computer that is connected to Plastic Cloud, so it caches data when working directly connected to Cloud to avoid redownloading the same data on branch switches.

It won't be like the actual proxy server in terms of deployment, since it will have to be totally automatic and with zero maintenance. It will have at least a recently used list of blobs to auto-remove the least used.

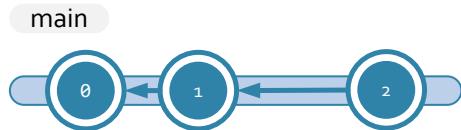
The goal is to dramatically speed up updates by removing the chance to download a file more than once from a distant server.

How replication works - push/pull

After looking at what working distributed is, and the layouts supported by Plastic in terms of combining distributed, centralized and even implementing multi-site, it is now time to look at how distributed actually works and what the push and pull operations do.

Globally unique changeset numbers

When you look in a Plastic repository, you'll see that changesets are numbered as follows:



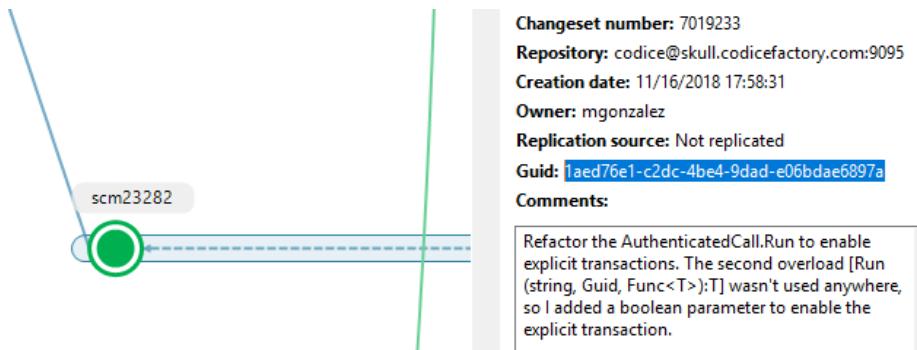
There is always a *changeset zero* automatically created together with the main branch when you create a new repo.

Then, as soon as your checkin changes, there will be other changesets, in my example 1 and 2.

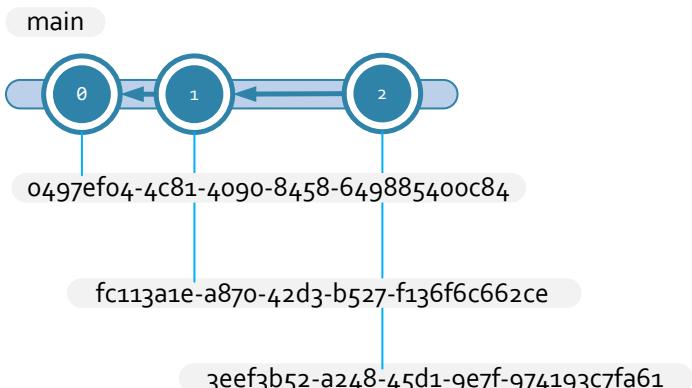
But, these changeset numbers are local to the repository. I mean, they are good if you work centralized, they are meaningful and easy to remember numbers.

But as soon as you work distributed, local numbers are not good anymore and you need Globally Unique Identifiers – GUIDs.

So, if you look carefully in your Plastic SCM Branch Explorer you'll see there is always a GUID number for each changeset.

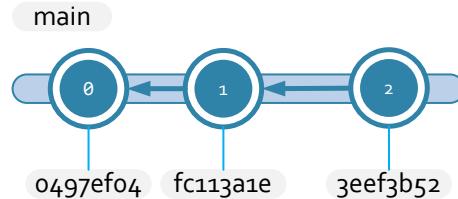


Thus, changesets are identified by two numbers: locally by their changeset number, but globally by their changeset GUID.



The GUID is a long identifier, so for short I'll be referring to it in a shortened way just by its first section. This shorter section is not guaranteed to be unique, but it is normally good enough to refer to it in examples. And chances are, you can even use it to refer to a changeset globally to a team member, since chances of a collision are small.

Considering that, the previous diagram could be simplified as follows:



How push works

A push takes changes from one repository and pushes them to a different one. The repos can be on the same server but they are typically on separate ones.

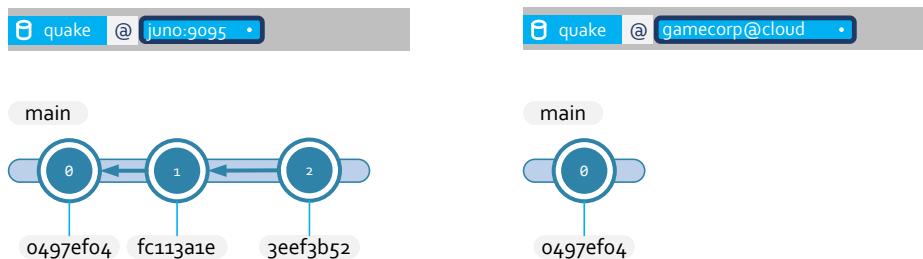
Let's follow a simple example step-by-step so we can ensure you get a solid understanding of how it works.

First, I start with a repo created on a server call `juno`. In fact, `juno` used to be our on-premises main server years ago 😊. By the way, instead of `juno`, it could be a local server, something like `quake@local` for Cloud Edition users, or `quake@localhost:8087` if you have a local Team server on your machine. The actual operation will be exactly the same.

As you can see, my `quake` repo has three changesets.

There is a different repo hosted in my Cloud organization: `quake@gamecorp@cloud`. Again, instead of a hosted repo in Plastic Cloud, it could be something like `quake@skull:9095` for an on-premises server.

Notice how the changeset `0` has the same GUID in both repos. This is on purpose. When we designed Plastic, we wanted it so that all repos had a *common ground* so they could all replicate with each other. More on this in a second.

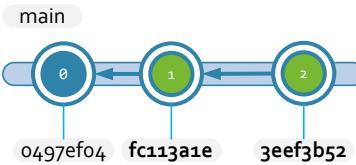


Then, what if you now decide to push from `main@quake@juno:9095` into `quake@gamecorp@cloud`?

The first thing Plastic has to do is to determine what needs to be pushed.

To do that, server `juno:9095` connects to `gamecorp@cloud` and retrieves its changesets on the `main` branch. Then, it finds there is a common point: the changeset `0`, and that the two "children" of `0` —`fc113a1e` and `3eef3b52`— are the ones that need to be pushed to complete the replica.

quake @ juno:9095 •



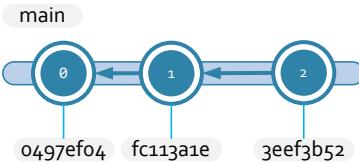
quake @ gamecorp@cloud •



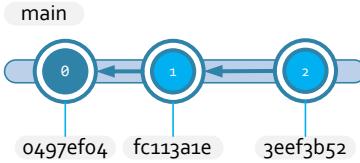
Then, the *source server juno:9095* (source of the push) will package the two changesets and send them to *gamecorp@cloud*, together with its corresponding data. I mean, it will send all the needed data and metadata.

Once the operation completes, the situation will be as follows:

quake @ juno:9095 •



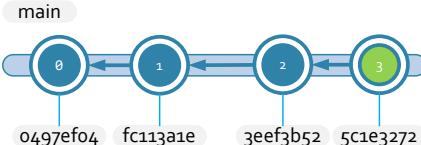
quake @ gamecorp@cloud •



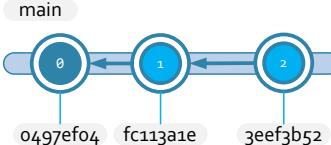
At this point, the two repos will be identical.

Now, what happens if you checkin a new change to *quake@juno:9095* and want to push again? Well, the same calculation will happen again:

quake @ juno:9095 •



quake @ gamecorp@cloud •



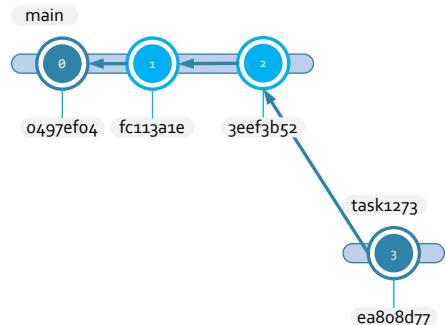
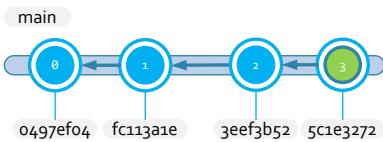
This time, the server *juno:9095* determines that *5c1e3272* will be the changeset that needs to be pushed, because during the negotiation, it found that the remote server already has *3eef3b52*.

And this is how push works. As you can see, it is not rocket science.

Why normal changeset numbers don't always match and we need GUIDs?

Let me go back again to the changeset numbering problem and the GUIDs. Let's start with the example above: there is a new change *5c1e3272* to push between *juno* and *gamecorp@cloud*. But, something happened in *gamecorp@cloud* meanwhile: a new branch was created there.

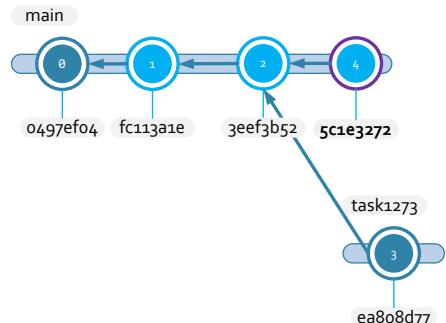
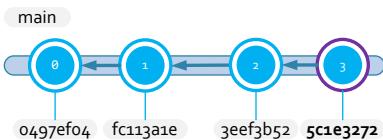
Please also note how I'm starting to color changesets based on their origin: one color for the ones created at *juno*, a different one for the ones originally created at *gamecorp@cloud*.



The new changeset on branch `task1273` at `gamecorp@cloud` has changeset number `3`. It collides with the changeset `3` created on `juno`, although they are obviously different.

Locally, each repo tries to follow a numbering sequence, and that's why numbers can collide.

Check how the two repos look like once the new push operation is completed:



Changeset `5c1e3272` is local number `3` in `juno`, but number `4` in `gamecorp@cloud`.

That's why you can only trust changeset numbers locally, but need to use GUIDs if you work distributed.

I hope this explanation makes it super clear why we use the two identifiers.

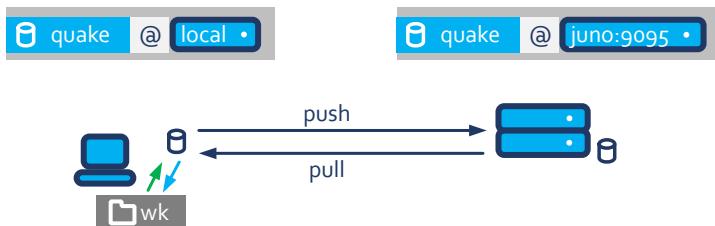
Push vs. pull

I'm going to explain what is the difference between a push and a pull operation in Plastic.

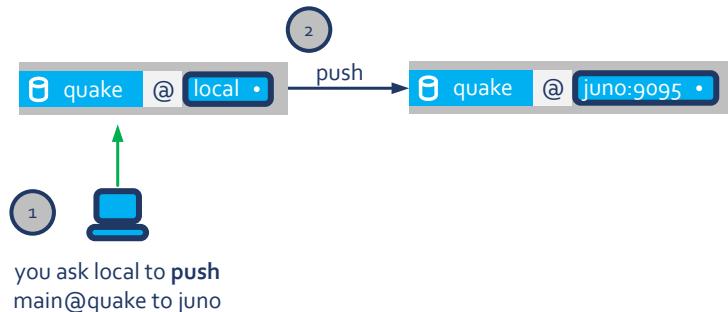
The obvious is true: push means sending changes from A to B, pull the opposite, bringing changes from B to A.

But, there is a small detail to take into account since in Plastic you are not restricted to push/pull only repos hosted locally, but also repos hosted on different servers.

The following is a typical push/pull schema between a local repo (hosted on the server called `local`) and an on-premises server called `juno`.



When you ask your **local** server to push to **juno**, basically what happens is something as follows:



Note for Git users on GUIDs vs. SHAs

If you have a Git background you might question why we don't simply do like Git and keep a single identifier.

The answer is a short story that goes back to our beginnings. When we started, our goal was to replace the Subversions and Clearcases of the world, and Git didn't even exist when the first designs of Plastic were put together. In all version controls prior to Git, changeset numbers were readable by humans. Of course, a GUID would be better in terms of implementation, but we didn't want our users to shout us if we asked them to handle such a long set of numbers and letters.

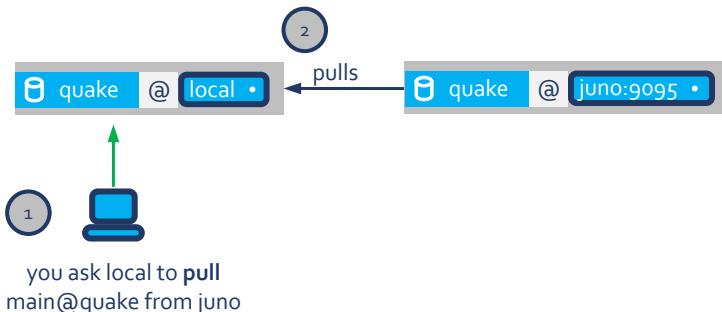
Years later, Git became the most extended version control and developers accepted long SHAs as identifiers happily. (Well, probably not so happily but they had to accept it anyway).

Long unreadable identifiers became normal, so Plastic GUIDs were now perfectly fine for everyone.

But, we still kept the *local* human readable changeset ids. One reason was backwards compatibility, the other that many teams were not really using distributed on a daily basis so they were much happier with their good-old sequences.

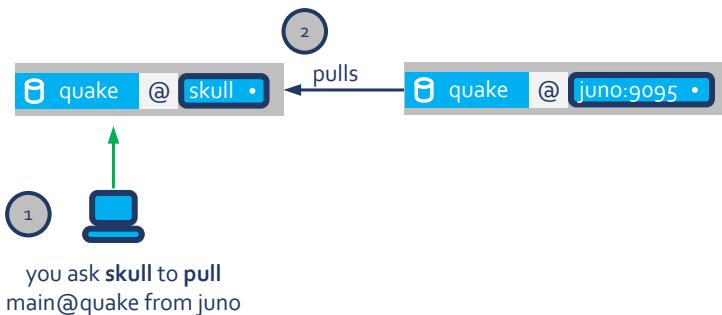
Your Plastic *client* connects to the **local** server and asks it to connect to **juno** and negotiate the push. Then the steps described in [how push works](#) happen.

Now, when you pull from **juno**, you are asking the local server to get changes from **juno**. Again, your client connects to the **local** server, then the **local** server connects to **juno** and negotiates the pull. The pull is quite similar to the push but instead of sending changes it simply performs a negotiation to *download* data and metadata.



But, remember that in Plastic you can perform push and pull operations not only from local repos, like happens in Git, but also connecting to remote repos.

Thus, the following diagram shows a pull operation but this time pulling changes from **juno** into a new server called **skull**.



You first ask **skull** to pull from **juno**. Then **skull** connects to **juno** and negotiates the pull, and then downloads the metadata and the corresponding data.

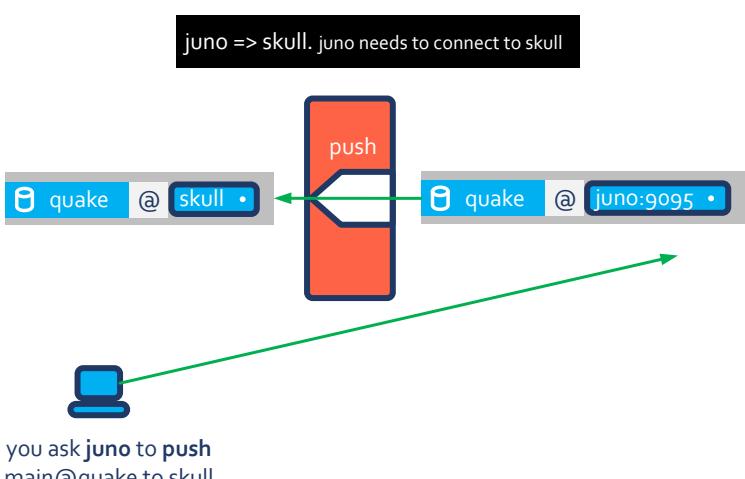
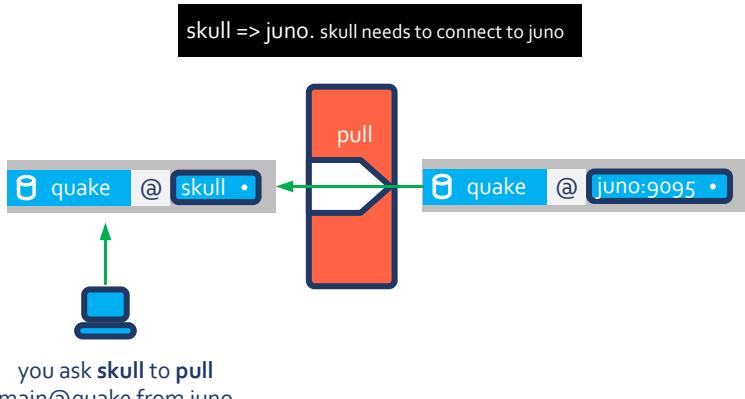
Think for a moment about what I described. Unlike what happens in Git, you can trigger *replicas* from any server, not only from your local repos.

Then, in Plastic you can achieve the same behavior in different ways: pull from **juno** into **skull** is equivalent to a push from **juno** to **skull**.

The following diagrams illustrate the equivalent scenarios and highlight a key point: the operations are equivalent but the required network connections are not.

When you pull from **juno** into **skull**, **skull** is the one connecting to **juno**. The diagram shows the direction of the connection, and then how the data is transferred.

Pull **skull** <- **juno** is equivalent to push **juno** -> **skull**
but the network connections required are different

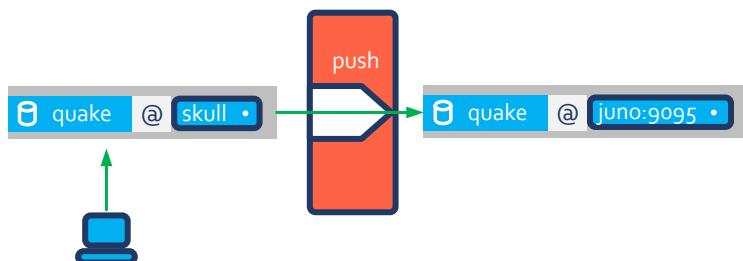


The important thing to note here is that while the two previous operations are equivalent in terms of data transfer, the required network connections are different. **skull** might be able to connect to **juno**, but the opposite might not be true, so the second alternative, pushing from **juno** to **skull** might be forbidden due to network restrictions.

The next diagram explores a new *equivalence*. A push from **skull** to **juno** is operationally the same as a pull made by **juno** connecting to **skull**. Again, the network connections are different and the restrictions might disallow some combinations.

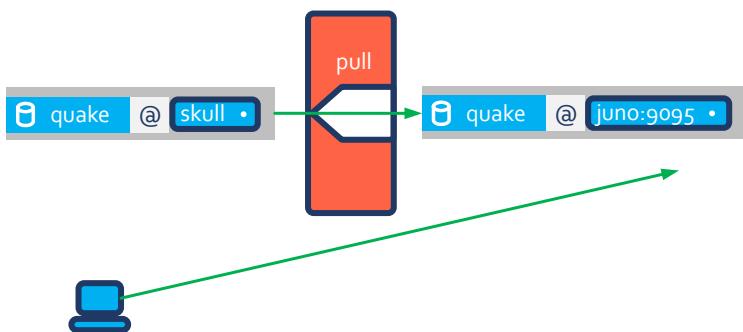
Push skull -> juno is equivalent to pull juno <- skull,
but the network connections required are different

skull => juno. skull needs to connect to juno



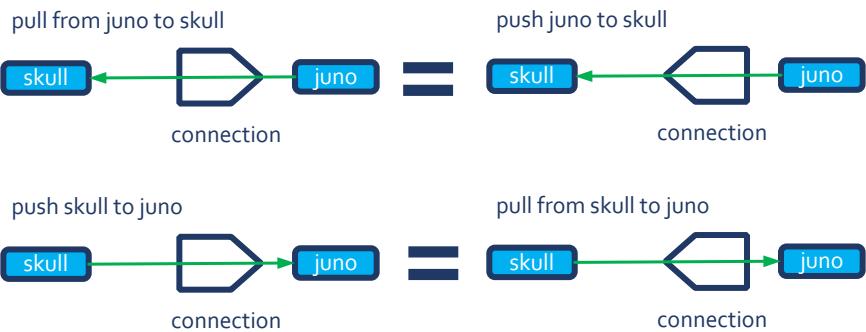
you ask **skull** to **push**
main@quake to juno

juno => skull. juno needs to connect to skull

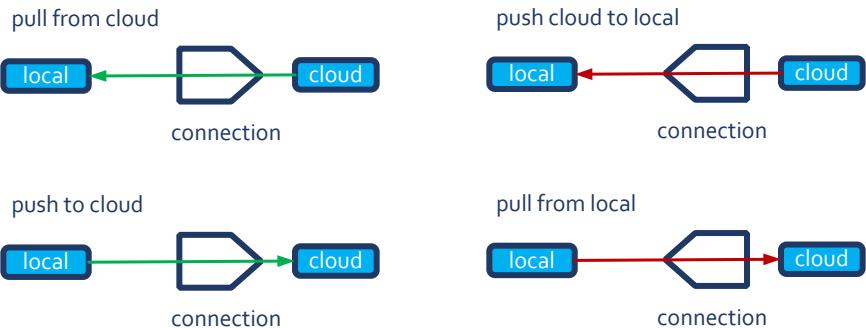


you ask **juno** to **pull**
main@quake from skull

I hopefully made this crystal clear. It is not difficult at all, but very often we find users being confused about the equivalences of push and pull between servers. That's why I created a new summary diagram:



Finally, for the users that are going to use Cloud Edition or Cloud extension, look at the following:



- You can pull changes from Cloud, but Cloud can't connect to your local server to push changes directly to your local repo.
- You can push from your local server to Cloud, but definitely your local repo won't be accessible from cloud, so it can't pull from you.

Firewall configuration

The direction of the required connections described above is important for the firewall configurations. Normally, local repos won't be accessible for security reasons, so they'll need to push/pull from a remote, but a remote won't be able to push/pull from them.

The firewall configuration becomes important in multi-site scenarios, depending on the types of operations you need to achieve.

It is not difficult at all to setup, but simply keep in mind the diagrams in the previous section so you have a crystal clear understanding of the connections required.

I know this might be obvious for most of you, but just wanted to ensure that it doesn't end up becoming a source of frustration 😊

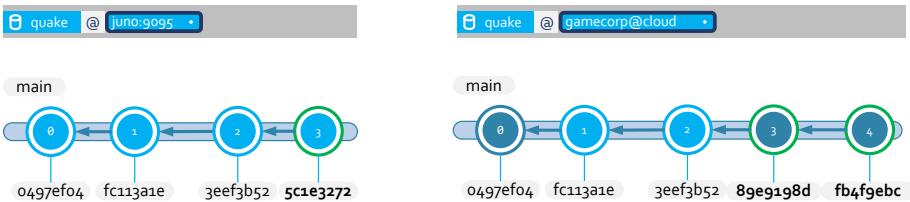
Handling concurrent changes on replicated repos

All the examples I described so far were pretty easy because our imaginary developers were kind enough to make changes only on one of the replicated repos at a time, so there weren't concurrent changes and hence no need for conflict resolution.

Concurrent changesets on different repos

But now, it's time to find out how to handle changes that happen concurrently. Let's start with the following scenario. We originally started with a quake repo in [juno](#) that we pushed to [gamecorp@cloud](#). Now, developers created changesets concurrently at both sides. As you can see, there is a new changeset [5c1e3272](#) on [quake@juno](#) and two new ones at [gamecorp@cloud](#).

- changeset created in juno
- changeset created in cloud
- newly created cset

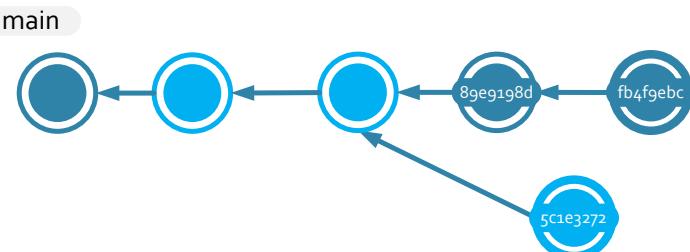


If you try to push from [juno](#) to cloud now, Plastic will complain saying the push can't be completed because it would create a *multi-head* on destination.

A multi-head? Is that some sort of monster??

Multi-head explained

If the push was allowed, your cloud repo would end up like this:



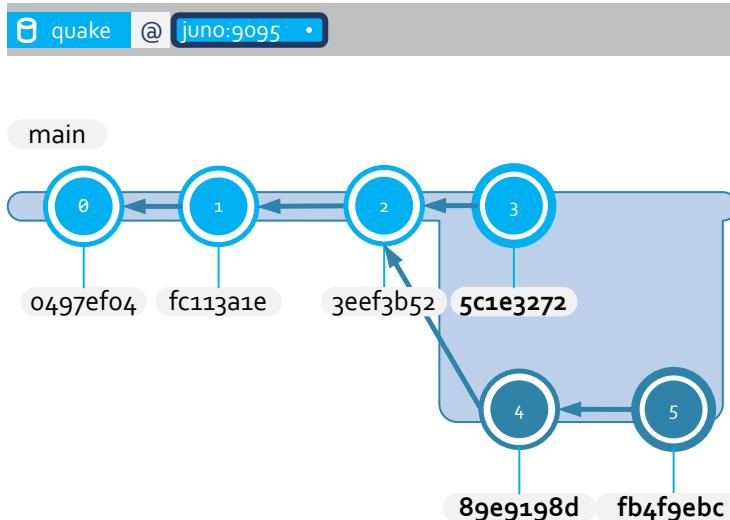
Where both [fb4f9ebc](#) and [5c1e3272](#) would be heads. If that happens, which would be the changeset at the head of the branch? What would be downloaded when someone tries to update latest from main? Which changeset would be the selected one?

That's why Plastic prevents the creation of multi-heads on push, because it asks you to fix that situation

in your *source* repo before you push.

Use pull to resolve concurrent changes on your repo

That's why you will need to pull from Cloud before pushing your changes. Once you pull, the repo at [juno](#) will be as follows:



Basically, the two changesets created on Cloud will be downloaded and linked correctly. [89e9198d](#) as a child of [3eeef3b52](#).

The problem is that you are again in a multi-head scenario. In reality, the old head cset number [3](#) ([5c1e3272](#)) will still be the official head of the branch [main](#) at [juno](#), but if you try to push now to Cloud, Plastic won't let you complete the operation warning you that you have two heads.

Update will still work correctly downloading cset [3](#).



Note on cset numbering

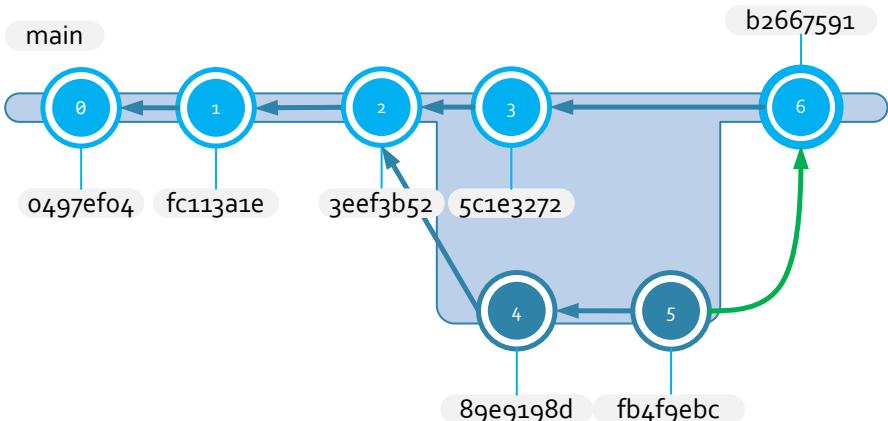
You can see how local changeset numbers (the integers) don't preserve the original numbering. [89e9198d](#) was number [3](#) on Cloud but it is renumbered as [4](#) in [juno](#). What stays constant is the GUID.

Now, what is the solution to this situation? We need to merge [3](#) and [5](#) to create a new changeset that will be the single head of [main](#) on [juno](#) and then we'll be allowed to push.

The figure shows how the new changeset [b2667591](#) combines the two old heads and creates a single unified head. Now the branch is ready to be pushed to Cloud.

quake

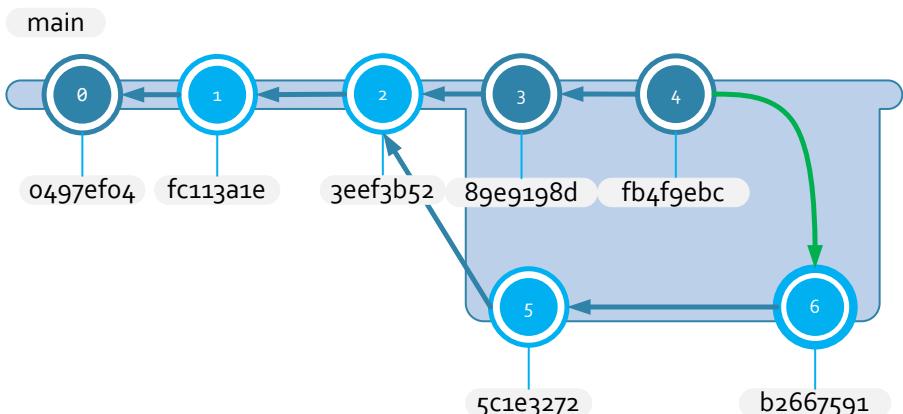
@ juno:9095



Once the push to Cloud is complete, the result will be as follows:

quake

@ gamecorp@cloud



As you can see, the final result, changeset **b2667591** means at this point the two repositories are equivalent, although their evolution was different.



I drew the *Branch Explorer* differently on Cloud compared to juno to actually explain how it works. But, the Branch Explorer in the GUI will always place the changeset that is head of the branch on the principal *subbranch* so the real diagram will be exactly the same on both Cloud and juno: the one I drew for juno above.

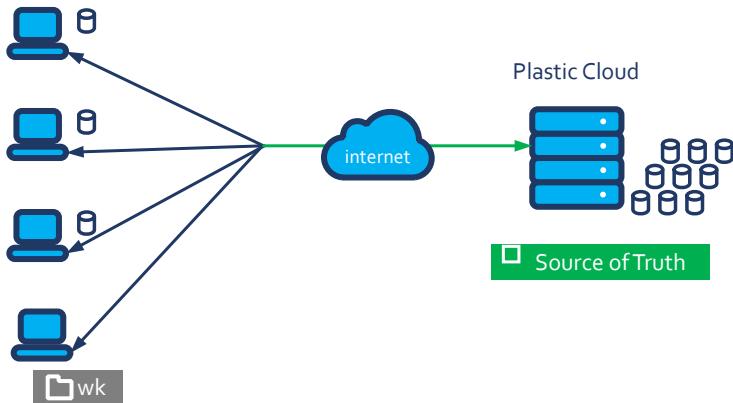


We also introduced the concept of *subbranches*: they are like branches inside a branch. In reality, changesets form trees, and while we expect a branch to be just a line, it can be a tree on its own.

Sources of truth in distributed development

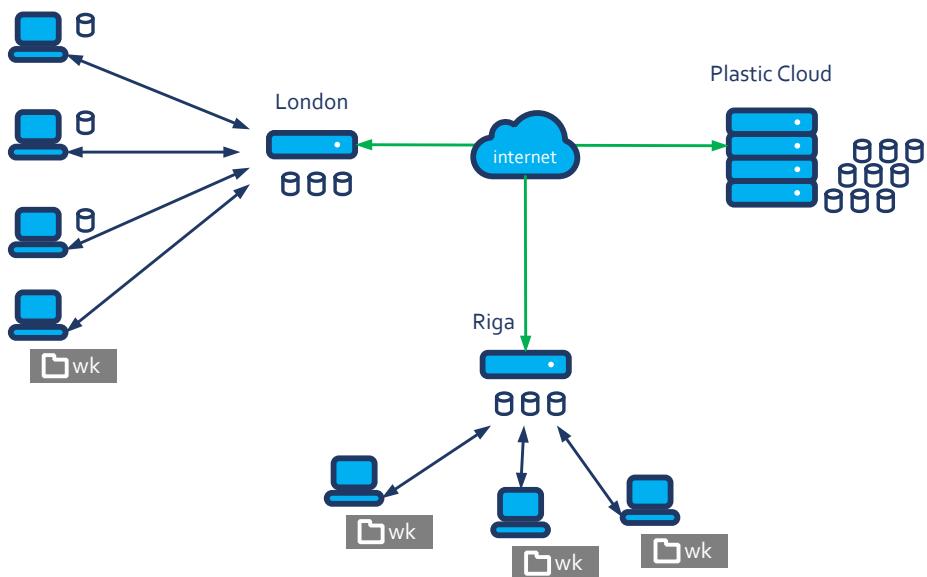
Single source of truth

When you work with a single central server, it is obvious which one is the source of truth: the repos on the server. Even if everybody has their own repo with a clone or a partial replica of one of the repo and work in push/pull, everyone knows where the master is.



This is basically the model that most of the DVCS follow. In fact, this is what GitHub is all about: a central solution for a distributed version control (beyond many other things like an awesome social network for programmers).

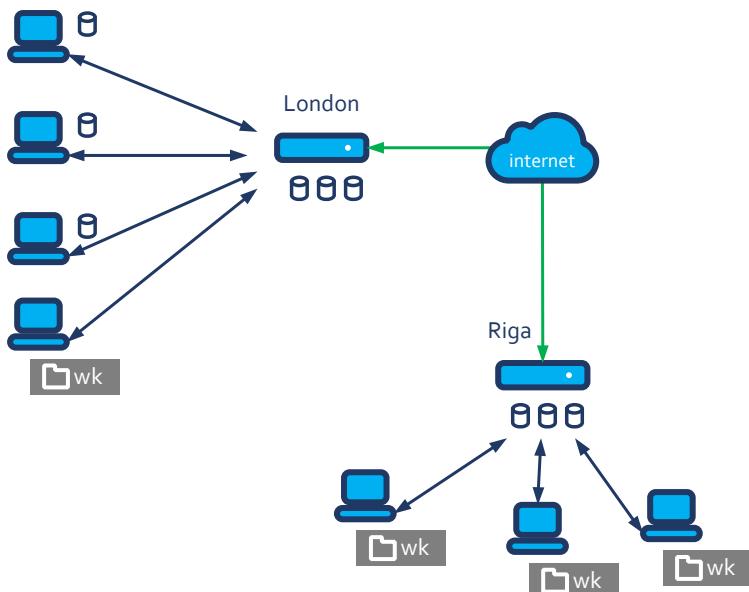
Now, if we go for a slightly more complex deployment, with several sites collaborating with each other, is it still easy to know where the source of truth is?



In this scenario, while there are two sites with their own servers, it might be easy to propose the Cloud as the rendezvous point where all changesets and branches are pushed. It might perfectly be the single source of truth.

Shared sources of truth

Now, what if we remove Cloud from the picture in the previous scenario? The result would be as follows:



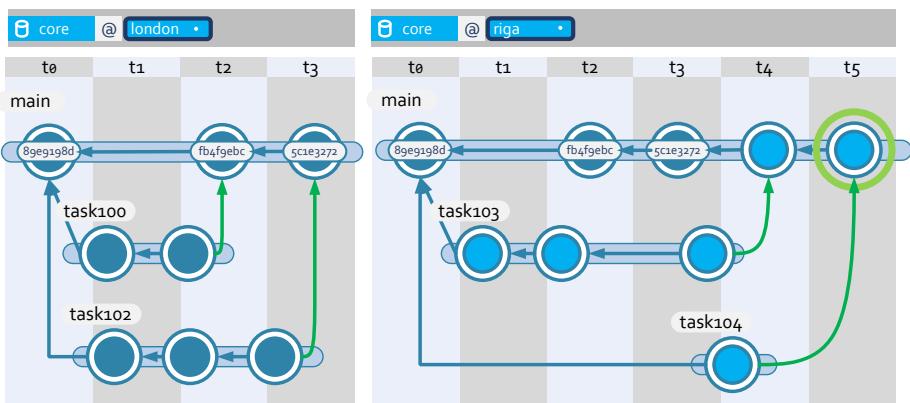
Now, it is not so clear anymore which one is the leader and which one is the follower.

Of course, it can be immediately resolved by a company-wide agreement: **Riga** is the main server, everything must be pushed to **Riga**. Very often this doesn't even happen by consensus but due to historical reasons: the original site rules.

The alternative is a shared ownership of the *truth* that can end up being a very good solution too: **London** and **Riga** can share the main versions, but probably there is no need for both to contain all the task branches created at each site. Each one can be the owner of certain repos. But let's focus on a single one: how is it possible for them to collaborate without both sides being some sort of perfect mirror of the other?

Look at the following diagram:

- changeset created in **Riga**
- changeset created in **London**



time	London	Riga
t0	The initial changeset was created on t-zero in London and later pushed to Riga	
t1	One developer starts working on task100 , and a second one on task102	First changeset of task103
t2	Second changeset of task100 and task102 . task100 is merged to main .	Second changeset of task103
t3	task102 is finished and merged to main	task103 is finished
t4	main pushed to Riga	Changesets fb4f9ebc and 5c1e3272 are pushed by London and appear at t2 and t3 (when they were originally created) task104 is finished task103 is merged to main ⇒ merged with the changes coming from London
t5		task104 merged to main and a new label created ⇒ new release

What you can see from the example is:

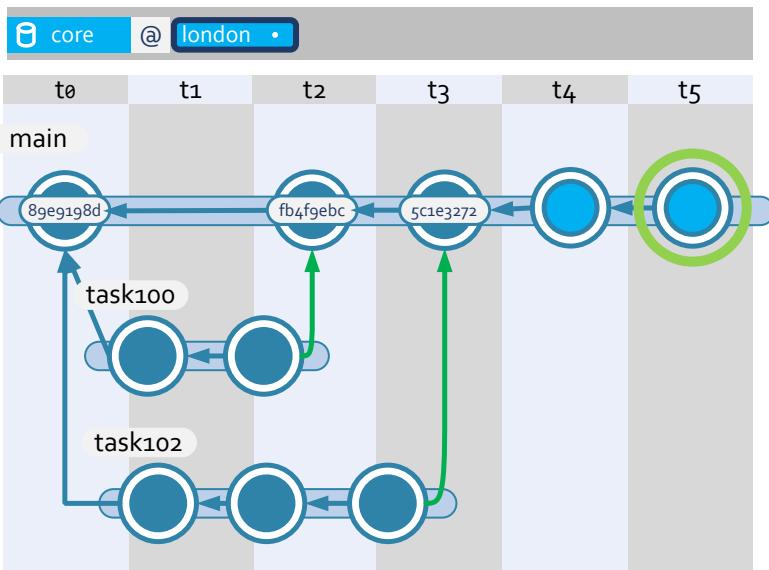
- **Riga** is the main site where new versions are created. That's why **London** pushes **main** to **Riga** once

branches are merged.

- It is too complicated to create releases for the same repo at two sites, for practical reasons: versions could definitely collide. I mean, it is doable with some sort of coordination to create new version numbers, but it would be a little impractical probably (or at least a good topic that is worth further discussion).
- London has some branch testing in place to decide if a branch is ready to be merged.

I was super careful to merge branches in Riga only after the changes from London were received. This is not realistic, but I did it to keep the scenario simple. Do you see what would have happened otherwise? If new changes were on main in Riga in t2, for instance, then London would have required a pull and merge (and subbranch) before pushing. Not hard, but it would be more complex than the scenario I drew.

Now, what happens in London once a new version is created in Riga? It just needs to pull main:



There are a few interesting takeaways here:

- The core repos at London and Riga are equivalent although they are not exact clones. I mean, the new labeled changeset at both sites is a snapshot of the exact same project tree. You can create two workspaces, switch each to the same changeset on the two different repos, and they will contain exactly the same content.
- But, the two repos are not exact clones, because they do not contain the same branches. task100 and task102 are missing in Riga. Yes, their resulting changes are there in main, but the branches are not there. Likewise, task103 and task104 are not in London.



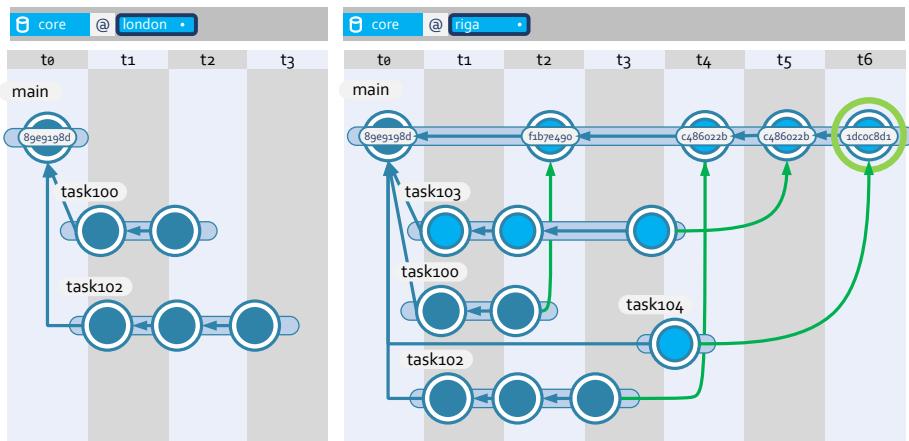
I'm using a multi-site naming convention, naming my servers as London and Riga, which implicitly says they are main servers on two different sites. But the entire reasoning stays true if instead of London and Riga they are your local and Cloud, or maybe your local and your team's on-premises central server. And even your repo and your colleague's repo.

Our recommended option – single source of truth and not exact clones

Once I introduced a couple of choices and the concept of different repos not having to be exact clones, I'm going to share what we think makes most sense.

To me, the previous scenario would have been much simpler this way:

- changeset created in riga
- changeset created in london



The workflow is not very simple:

- **London** never merges branches to **main**. It can create as many task branches as needed, but they will be pushed to **Riga** once they are finished (or as many times as required during the process, no need to delay the push until the task is done).
- Finally, all branches are at **Riga** as if they were created there locally (well, in fact, Plastic always keeps track of where each changeset was created). The following screenshot shows a changeset in our repo **codice@skull** where its replication source (where it was pulled from) is a different server. As you can see, Plastic keeps track of that. Among other more important tracking issues, this info lets you color the changesets in the Branch Explorer depending on their origin.

Changeset number: 7019504

Repository: codice@skull.codicefactory.com:9095

Creation date: 11/23/2018 15:23:36

Owner: bruiz@codicesoftware.com

Replication source: codice@ssl://codice@cloud.plasticscm.com:8086

Guid: a42b2d55-ec3d-46a4-9878-796752a6e1b0

Comments:

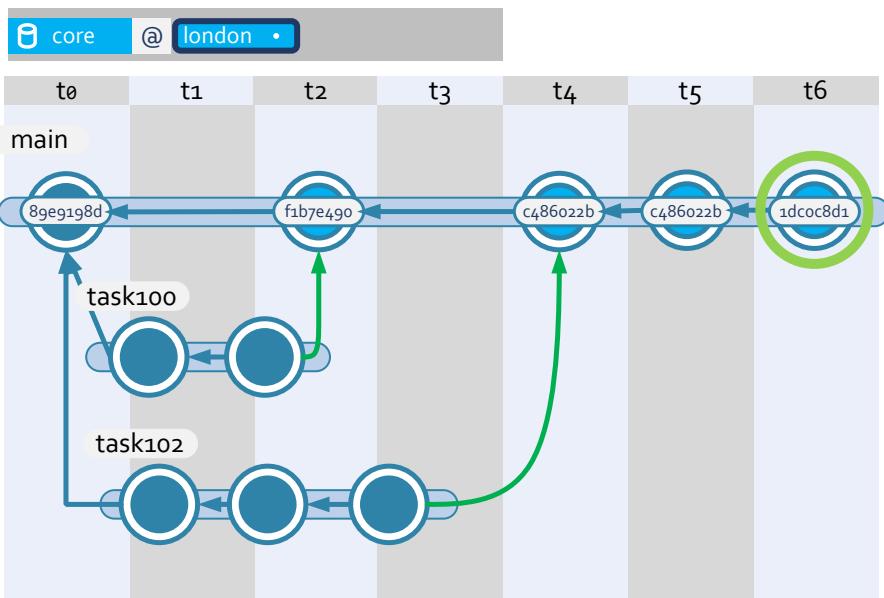
Add a test to reproduce the issue.

The test opens the file and keep it open in the same way than

- **Riga** is the one doing all merges. It makes a lot of sense because this way you have a CI system in place just monitoring the **core** repo at **Riga**.
- The decision to create a new version happens based on the info in **core@riga**, which greatly simplifies things. Whether you are using a mergebot or triggering new versions manually, having this *single source of truth* makes things simpler.
- **Riga**, as the container of the entire project, is the key one to backup and protect in terms of data loss.

Let's see now what happens to **London** once the new version is created: it can pull **main** from **Riga** to get the new version, and the merges of the **task100** and **task102** branches. Notice how the other branches **task103** and **task104** do not need to be pulled from **Riga** at all, and the **main** branch will still be consistent and equivalent to **main@core@riga**.

- changeset created in **riga**
- changeset created in **london**



This scenario I just described can happen in a multi-site situation like the one I also described before. But it will also be extremely common with developers working distributed in their own repos. Let me explain:

- As a project contributor you pull **main** from **Riga**. No need to download the complete history of the project. It would consume much more disk space and time, for no extra benefit.
- You will create as many branches as you want locally. Normally they'll be *your* task branches, the ones you worked on.
- You will push your branches back to **Riga** as often as you need, and surely once you have finished your work. Most likely, you'll mark your finished task branch with an attribute, that will be also replicated to **Riga**.
- You'll regularly pull **main** again to get newer versions and start your new tasks from there.

- Finally, your repo will contain much fewer branches than [Riga](#), but that's not an issue at all.



Something very important to keep in mind is that if your repo doesn't have all the branches, it doesn't have the entire merge tracking. What does this mean? Well, you are safe to merge between branches you created, but if your repo is incomplete in terms of history, it might get different merge results than if the merge is performed in [Riga](#) where all the branches are present. I'll cover this in greater detail at "[Merge tracking](#)".

Exact clones

What if you need to have two repos that are exact clones of each other? Is this achievable? And, I mean, exact clones, not just *equivalent* results as I mentioned a few times above.

As soon as developers work concurrently on two repos, there is no guarantee to have them as exact clones all the time. They'll be *eventually consistent clones* at the instant when all changesets are replicated to both sites and concurrent changes were merged and the results of the merges propagated. See what I mean? If you can stop working on both sides and just allow work to resolve conflicts on branches, then eventually the repos will be exact clones. But, since development normally never stops, then this eventual consistency tends to be delayed. It is not a big problem anyway, as soon as you are aware of it.

Now, there is an easier way to get exact clones without the need of dealing with concurrent changes: you just need to keep one of the repos read-only except for replicas coming from the source repo. If no actual development happens on the destination repo, then you can guarantee that they are exact clones, at least until the changeset that was just replicated into the destination.

This is very good if you need a failover repo (or set of repos) on a different server that you can switch to if something goes wrong on the main one, or to use it as a backup.

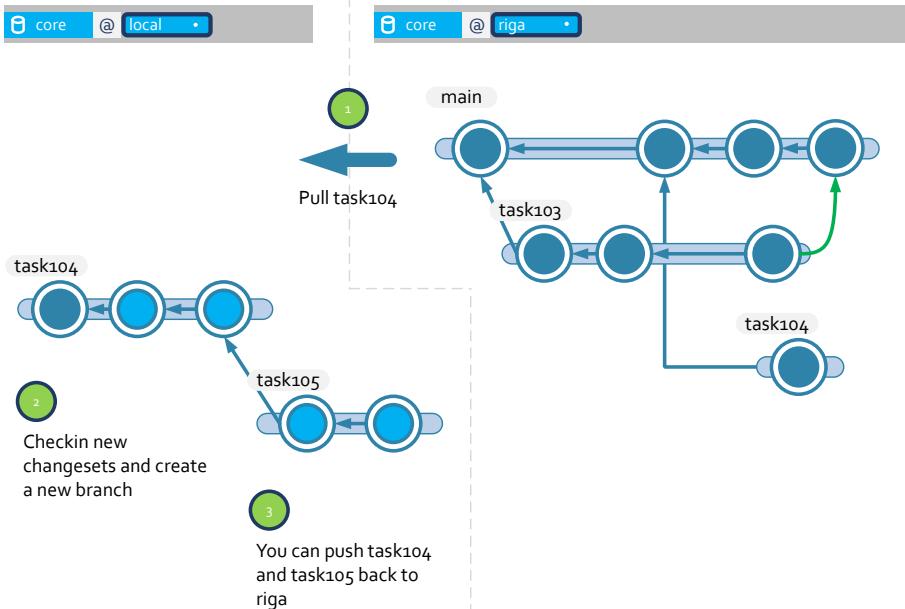
It is good because it can be fully automated without any manual intervention. I mean, if development happens at both repos, then there will be chances of manual conflict resolution during merges. If changes only happen at one side, then there will never be a need for manual intervention and replicas can be easily scripted. They can be launched by a trigger each time a new changeset is created on the source repo, or simply periodically every few minutes or hours, depending on your needs.

Partial replicas

All replicas in Plastic are partial. I mean, when you replicate a branch in Plastic you replicate just this branch, not the entire repo, unless you loop through all of them or create a sync view and push/pull the entire branch list.

Look at the following example:

- changesets created locally
- changeset created in riga



I can pull only **task104** to my local repo. There might be hundreds of branches in the central repo at **Riga**, but I'm only interested in branch **task104**. After pulling, I get a single changeset, but it is fully functional, it contains an entire copy of the working tree, so I can continue working from there creating more changesets and even new branches as the diagram above shows.

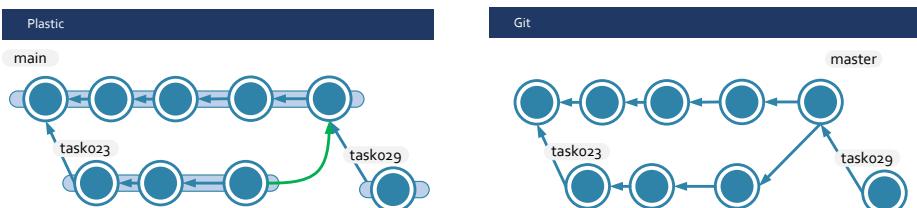
And, even more important: you can push both **task104** and **task105** back to **Riga** (or any other repo) if needed.

Replica in Plastic vs. Git

I like to compare the things that work differently in Plastic than Git since many of you will very likely have a Git background when you approach Plastic.

The replication process works differently in Git than Plastic. And these *partial replicas* are one of the key differences.

Let's go with a new example, this way checking two identical repos, in Plastic and Git:





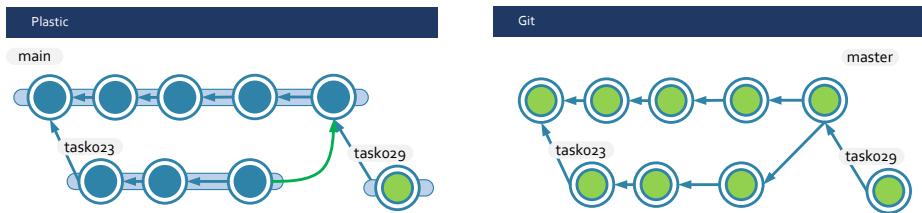
Note for gurus

Notice how in Plastic every single changeset belongs to a branch, there is a one-to-one relationship between cset and branch. While in Git, branches are just pointers and then a commit can be in more than one branch at a time.

So, what happens in each system when you decide to pull `task029` branch?

In Plastic, the two repos involved will negotiate which changesets need to be pulled and only the cset in `task029` will be selected. Of course, it doesn't mean only the files that were checkin in this cset will be downloaded, but I'll cover that later.

In Git, a tree walk operation will happen and considering the topology of the graph I drew, the commits marked in green will be downloaded:



In this case, Git would download the full repo, with all the changes. This is because the tree walk goes first to `master`, then the head of the `master` branch has two parents, so the walk will take both paths.

Of course, in Git there are ways to *limit depth* so you don't end up pulling the entire repo if you don't want to. But, when you limit depth, then there might be restrictions trying to push new commits back.

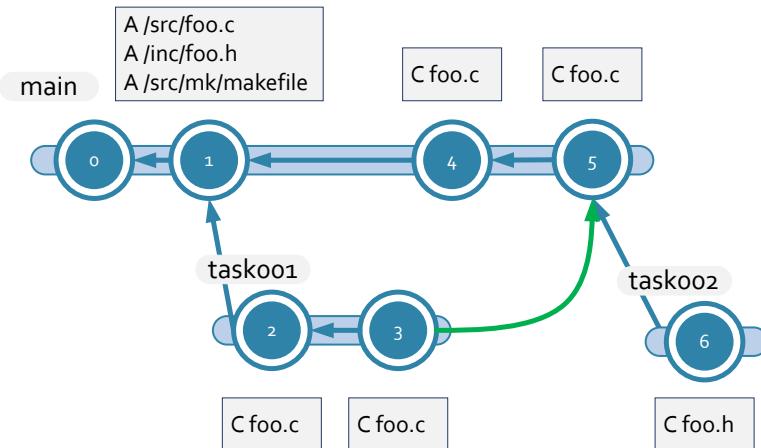
I wouldn't honestly say any of the alternatives is better than the other. Of course, I'm biased towards our design decision for obvious reasons, but what it is important to highlight the difference.

The dual lives of changesets

Changesets play a double role: they track the actual changes you checkin, and they are also a full snapshot of the entire repo at the instant of the checkin.

This is important for you to understand partial replicas.

As usual, I'm going to use a concrete example and a bunch of pictures to explain it.

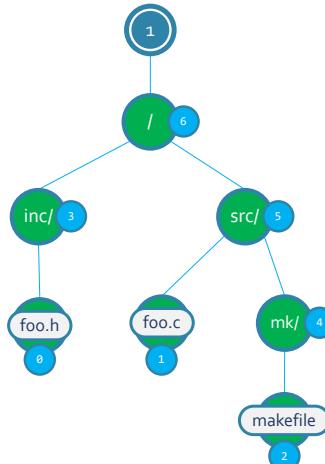


The changesets in the diagram are decorated with the actual changes.

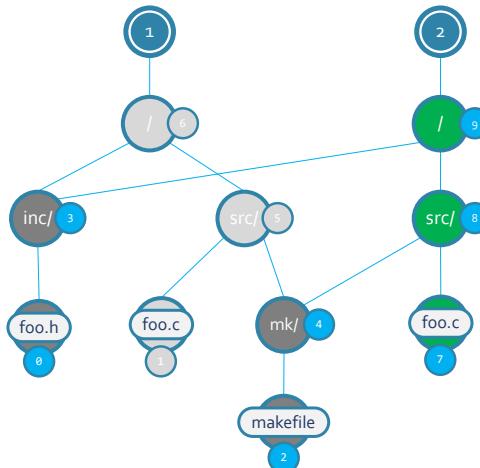
- First, I added three files to the repo.
- Then, I made a bunch of changes to `foo.c` both in `task001` and `main`.
- Later, I merged `task001` to `main`.
- And finally, I branched off `task002` and modified `foo.h` for the first time.

This explains what each changeset contains in terms of changes. But what are the actual trees they point to?

This is the tree for changeset 1, the one adding the three files. As you can see, the changeset points to a specific revision of the root directory which in turn points to the rest of revisions of the directories and files it contains.



Now, the changeset 2 in branch `task001` modifies `foo.c`. This is what its tree looks like:



This is how it works:

- `foo.c` is modified, so it goes from revision 1 to 7.
- But, to link to the new revision, directory `src` needs to go from 5 to 8 too, a new revision of the directory is created.
- The interesting thing here is that `src` revision 8 points to `foo.c` revision 7, but still points to `mk` revision 4, since the `mk` directory subtree didn't change. This is the actual magic that avoids recreating the entire tree on every single change.
- Going up in the tree, we need to create a new revision of `/` to point to the new revision of `src`. Of course, the `inc` subtree is not modified so `/` revision 9 links the previous `inc` revision 3.

In the figure above, green indicates the revisions that were created in changeset 1, dark grey indicates the revisions that are linked to the tree 2 but were not created there, and finally light grey indicates the revisions that are no longer pointed by changeset 2.



No drama

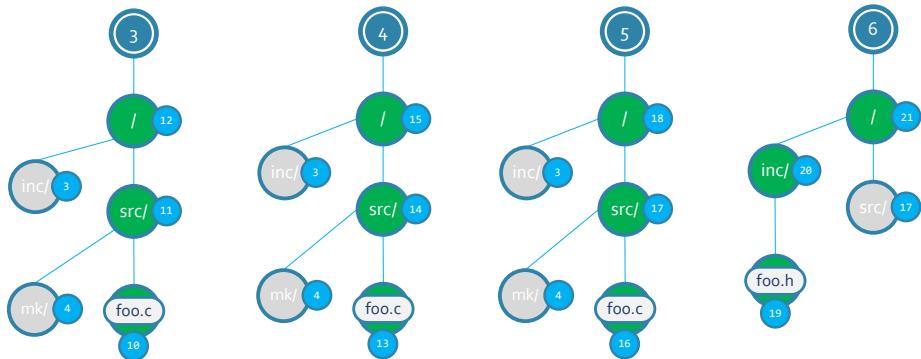
Well, I know this is not the most obvious concept in the book. Agreed. So many of you would say, well, this guy is trying to drive me crazy. I digress! It is way much simpler than anything you do on a daily basis as a programmer or designer!!! 😊



Key takeaway

Understanding how changesets actually work will turn you into a master of version control. So, I really believe this is 10 minutes really well invested 😊

The following graphic shows the trees of the next changesets: 3, 4, 5 and 6. Instead of drawing crossed lines all around, I just drew the references to the pre-existing subtrees in every new changeset tree.

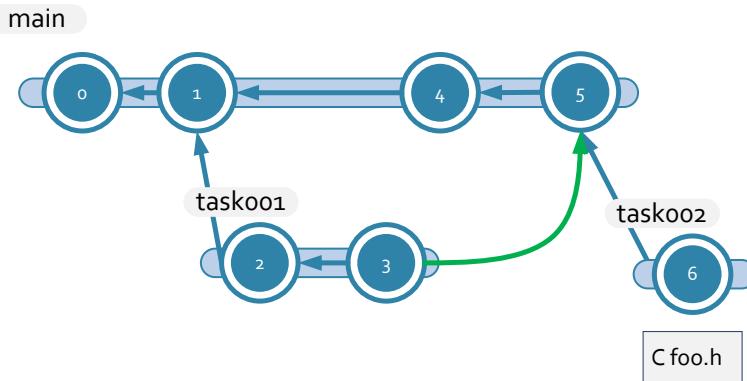


Revision numbers



Every single element (we call it an item) controlled by Plastic can be identified by the path inside its changeset, but it also has a unique revision number. Of course, the same revision number can be in many different changesets. This is the case of `inc` revision 3, which is used by changesets `0, 1, 2, 3, 4` and `5`. In contrast, `foo.c` has been modified many times, and there are many different versions of it. But `foo.c` revision 16 is linked by changesets `5` and `6`.

Now, let's go back to the replication scenario. When you pull `task002` you won't be just obtaining `foo.h` but the entire tree changeset `6` points to. It is an entire snapshot of the project, so it is fully operational.



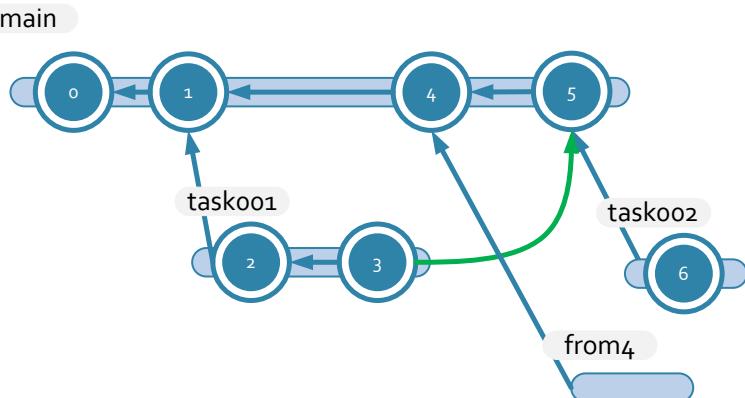
What you won't get if you just pull `task002` is the entire history of `foo.h`, `foo.c` and the rest of files. You'll just get the actual revision required to load the tree of changeset `6`.

In contrast, if you replicate `main` you'll be replicating all the changesets in the branch: `0, 1, 4` and `5`, and with them will come all the individual trees of these changesets, and a good part of the history of the files involved (excluding the changes made in `task001`, unless you decide to replicate that branch too).

A trick to replicate just a single changeset from main

Partial replicas allow us to replicate just one branch. But what if you want to replicate just one changeset of a branch and not the entire history? As you saw in the previous topic, it is technically possible.

At the time of writing this, we don't have an interface to specifically replicate only changeset `4` of `main`. This is not because of any technical restriction, we could easily do it, but we simply didn't add a way to specify just a single changeset in a pull.



Until we implement that, there is a trick you can use. Suppose you just want to replicate changeset 4 (and of course its entire working tree) instead of the entire `main` branch. Well, just create a branch `from4` as I did in the diagram, and replicate it. You'll get just the cset 4 replicated to your new repo.

Why this is useful? In our case, our main repo has been evolving since 2006. So, if I want to replicate `main` to my laptop, I'll be downloading a good number of gigabytes. That's not a problem at all, but suppose I want to save a bunch of gigs because my SSD is running out of space. Even if the repo is large, the working copy in the head of `main` is just 500 megs or so. I find it very useful to have the option to replicate just the equivalent of a working copy and be able to evolve from there locally, instead of having to download the whole history of the `main` branch.

I can always connect to the central repo to browse the entire history, while my local stays tiny.

Merge history and partial replicas

Like we mentioned while talking about "sources of truth", we must be careful when running certain merges on a partially replicated repo. This is because there might be some missing merge history that might alter the final merge result in the partial replica.

It is easier seen with an example:



If you merge **bug199** to **main** at **Riga**, the common ancestor for the merge will be the changeset marked as **B** in green. Which is the correct one.

Now, if you only pull **bug199** in your local repo and try to merge to **main** locally, the common ancestor will be changeset **A**, marked in red, and the result of the merge will be different (possibly more conflicts) than if run in **Riga**.

That's another reason why we prefer to do the merges on the *central* site.

Of course, you are free to merge locally between branches with a simpler hierarchy that you know you completely own, like a child branch you [created in your repo](https://www.plasticscm.com/download/help/partialreplica) [<https://www.plasticscm.com/download/help/partialreplica>].

Possible future change in merge tracking with distributed repos

For years we considered making a change in how merge tracking works with distributed repos. We consider 2 options:

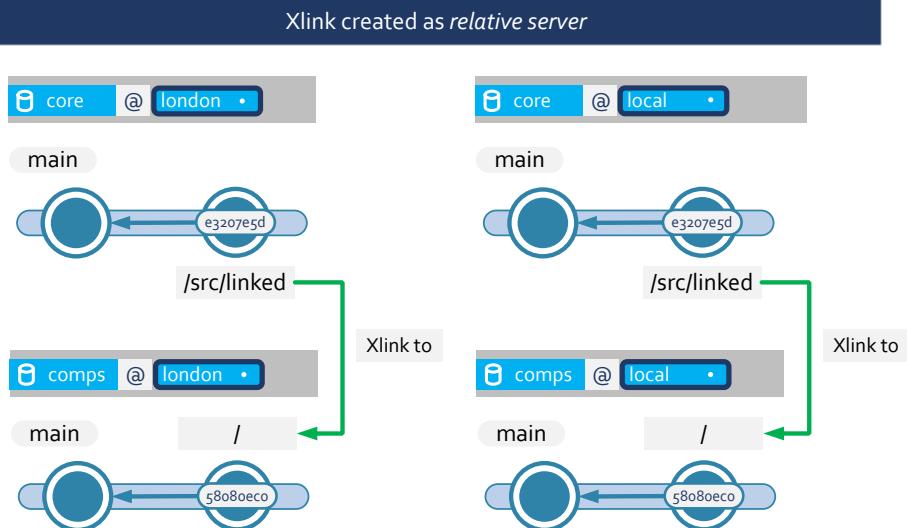
1. Ask the original repo for the missing merge tree information (when available), which sounds pretty cool under certain cases (especially for collocated teams with developers working in DVCS mode but with full access to the central repo).
2. Always replicate the entire changeset hierarchy (aka merge tree info), but not the associated data (so it would still be achieving the goal of partial replica but also detecting when intermediate csets are missing for merge).

We still haven't changed this behavior, but it is worth noting that this is something we have been considering for a while.

Xlinks with distributed repos

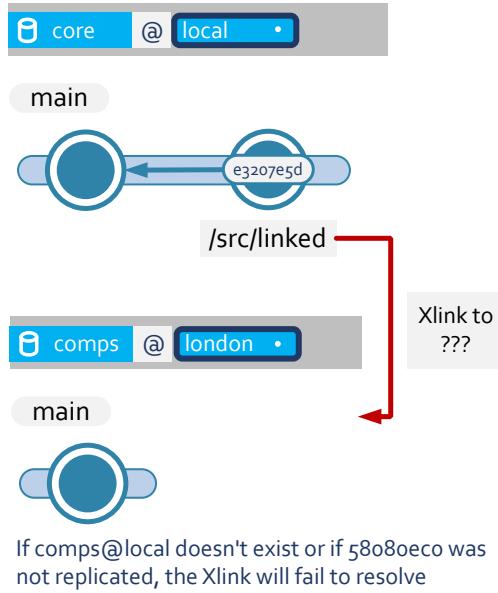
When you create an Xlink, there is an option in all GUIs and command line to specify if you want the Xlink to be relative to the server.

If relative server is set, it means that if `/src/linked` at repo `core@london` points to `/` at cset `58080ec0` at repo `comps@london`, if you replicate `core` to `core@local`, it will expect `comps@local` to exist too.



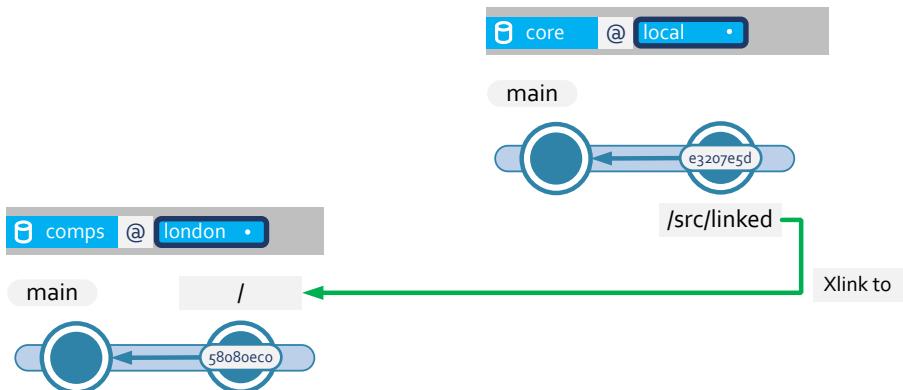
When you use *relative server* Xlinks you always need to remember to replicate the branches in the `comps` repo together with the ones in the `core` repo, otherwise the tree of the changeset will fail to resolve when you try to switch to it.

Xlink created as *relative server*



Finally, the other option is to use *absolute server* Xlinks, which means you want the xlinked tree to be obtained from the original server.

Xlink created as *absolute server*



It is a good option too, but of course it means you **always** have access to the `comps@london` repo (so your internet connection must be available) while working on `core@local`.

BRANCHING

Branches are central to Plastic. We already covered how important task branches are from a workflow point of view.

In this chapter, we are going to be more technical about what happens behind the scenes when you create a branch in Plastic.

Every repository starts with a main branch

Each new repository you create in Plastic, comes with a default branch called `main`. The main branch comes with a cset `0` that is just an empty changeset that simply contains the first revision of the root item `/` of the repo.

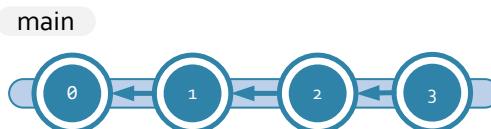


You can rename the `main` branch if you want, although most teams simply keep it and use it as a convention for the principal location to contain new versions.

Every changeset belongs to a branch

Even if you decide to skip branching entirely, you'll be using at least one branch, since every single checkin creates a changeset that belongs to a branch.

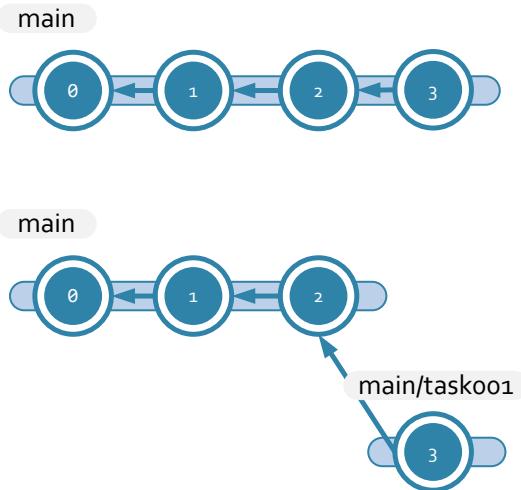
Every changeset has an associated task. Branches in Plastic are changeset containers.



As you can see, all four changesets in the figure belong to the branch `main`.

Creating branches is cheap

Look at the following figure:



Imagine I added 300k files to the repo in changeset 1 (three hundred thousand files). The cost of the two branch alternatives in the figure is exactly the same.

The only difference is the few extra bytes of metadata required for the branch object itself (name, owner, date, etc.). Creating a new branch is just adding a new light object to the metadata of the repo.

A branch is not a copy of the full directory structure, nor a special directory like it was in the days of Subversion or Perforce.

As you can see in the figure, every changeset points to its parent. This means it basically *inherits* everything and adds its own changes to the tree. So, if you modified three files in changeset 3, this is the actual cost of the changeset, independently of whether it was created on branch `main` or `main/task001`.

Every changeset points to a *given revision of the root directory* to create a snapshot of the repo. But, none of these trees are full copies, they are created incrementally starting from the previous tree and reusing most of it. For more information about how these snapshots work, see the section "[The dual lives of changesets](#)".

Branches have their own metadata

Unlike what happens in other version control systems, like Git, branches in Plastic are full entities, not just *pointers*.

This means every branch is decorated with specific metadata:

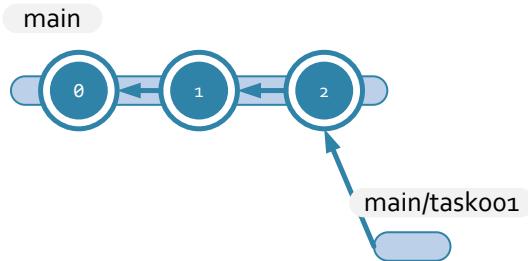
- Creation date
- Creator
- GUID
- Comments: Every branch can have a description, which is very useful to complement the name.

- Attributes: You can assign value-pair combinations to a branch. A typical use case is setting a status: resolved, open, merged, failed.

Branch hierarchies

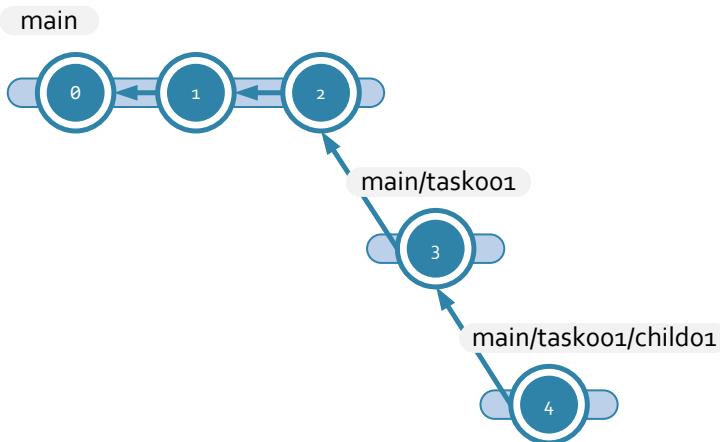
Child branches

Whenever you create a new branch, it is by default created as a *child* of the one it starts from.



In this case, `main/task001` is still empty (and in fact it points to the same tree as changeset 2), since it doesn't contain any changesets. Its name says that `task001` is a child of `main`. In fact, branch objects in Plastic have a *parent* field pointing to the branch they are child of.

Thanks to this parent relationship, it is possible to create entire branch hierarchies:

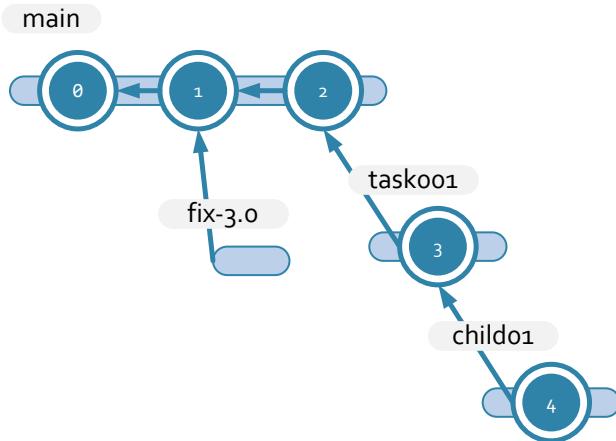


Branch hierarchies are a way to introduce a namespace structure in your branches. They don't have any other impact other than keeping the branch structure better organized.

Top level branches

In every repo, there is at least one top level branch: `main`. But, you can create as many top-level branches as you need. In fact, from a practical perspective, both child and top-level branches have the same behavior.

The following figure shows all branches created as top-level ones.



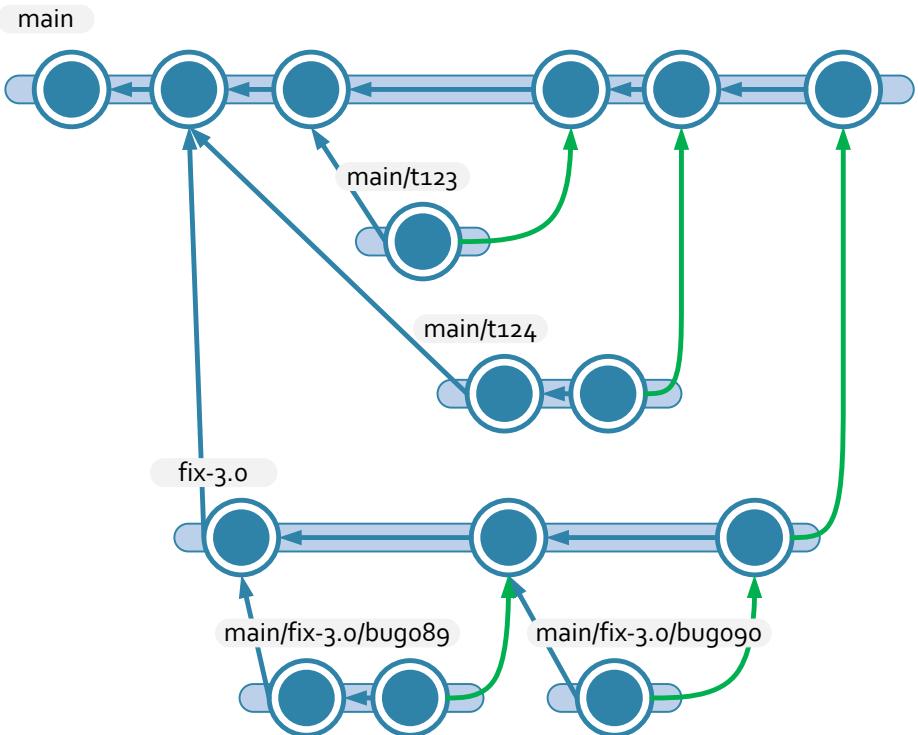
In many diagrams in the book I simply drew the branch name, not the full name including its parent. I mean, to keep the diagrams less verbose, I normally named branches as `task001` when in fact they should have been `main/task001` since, by default, Plastic creates branches as children of their parents.

A meaningful branch hierarchy

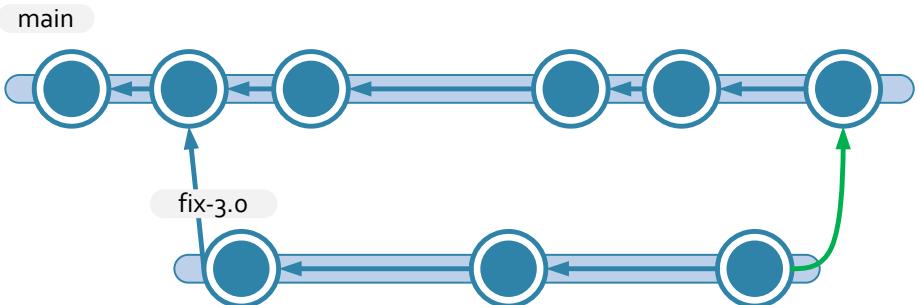
As I explained above, it is up to you to use top-level or child branches. Their only impact will be in how you name the branches and how it is easier for you to manage your project.

Let me propose a strategy that I find useful: keep top-level branches for *structure* branches and child branches for tasks and bug fixes.

Take a look at the following diagram. We have a `main` branch with a number of task branches. Then, a `fix-3.0` that can be a maintenance branch to evolve an older release, and it also has its own set of child branches.



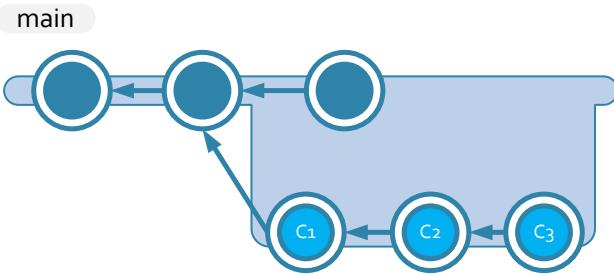
One of the advantages of defining a strict structure of top-level and child branches is that you can filter by level in the Branch Explorer in the Plastic GUI, and then easily focus on the structural view, by filtering out all second-level branches as follows:



This sort of simplification helps you focus on the structure of the repo by hiding more tactical short-lived branches.

Subbranches

Plastic can also manage *branches inside branches*. In reality, the changeset structure is the one creating the actual trees of divergent history. Following that train of thought, it is perfectly possible to have something like the following:



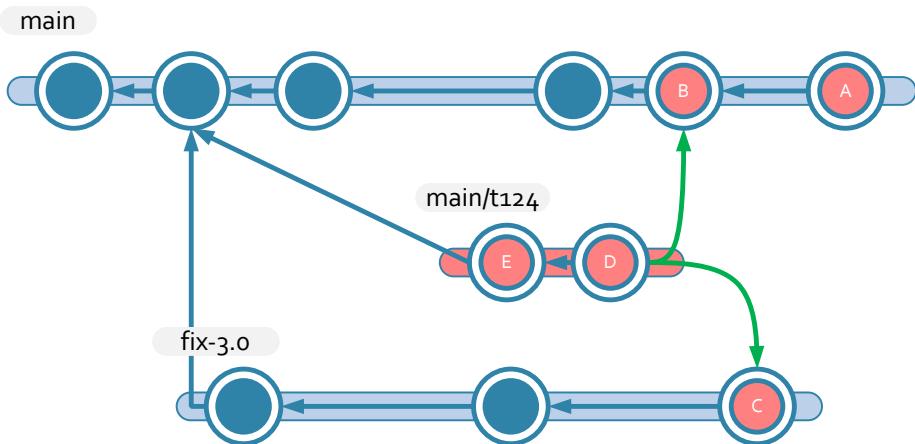
Changesets **C1**, **C2** and **C3** all belong to **main**, so they are a branch inside a branch.

To learn more about subbranches, see "[Handling concurrent changes on replicated repos](#)" because when creating subbranches, you typically need to make concurrent changes on the same branch in replicated repos.

Delete empty branches only

Branches can only be deleted when they are empty. This means that if you want to delete a branch, you have to delete its changesets first. A changeset can only be deleted if it doesn't have any descendants (children or linked by merges).

Consider the following scenario: If you want to delete **main/t124**, you'll need to delete its two changesets first. But, to delete changeset **D**, you'll need to first remove **A**, **B** and **C**, to get rid of all its descendants.



Yes, Plastic branches are admittedly hard to kill.

But, in our opinion, the question is: why would you want to delete a branch?

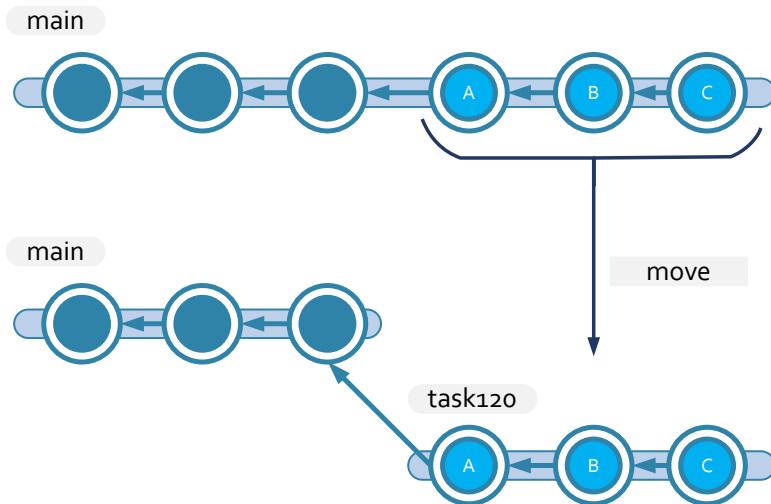
The answer is obvious for users coming from Git: deleting branches is a pretty common technique, you need to clean out old branches to keep the repo usable. But, that's not the case in Plastic. If you want to learn more about why we don't delete branches, please check "["We don't delete task branches"](#)".

Changesets can be moved to a different branch

Changesets belong to a branch, but they can be moved under certain circumstances.

A common example is that you mistakenly checkin changes to `main` instead of the right task branch because you forgot to create the branch and switch to it.

It is possible to fix the issue and move the changesets to the new branch:



Find the details about how to move changesets [here](https://www.plasticscm.com/download/help/movechangesets) [<https://www.plasticscm.com/download/help/movechangesets>].

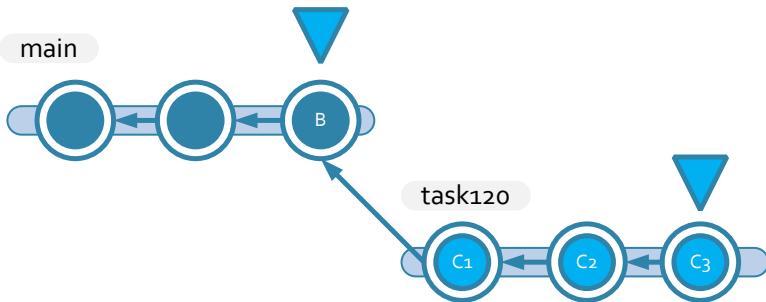
Diff branches

Branches are changeset containers, meant to hold parallel project evolution until divergent lines of evolution converge.

But, besides being containers and tenants of project hierarchies, branches also exist to be diffed.

This is very important to highlight because many users tend to think only in terms of changeset diffing and not full branch diffing.

What actually happens when you diff a branch? In the example below, diffing `task120` is the equivalent of diffing changesets `C3` and `B`.



$\text{Diff}(\text{task120}) = \text{Diff}(B, C3) = \text{changes made in } C1 + C2 + C3$

The important thing to keep in mind is that Plastic branches know where their starting point is, so when you diff a branch, Plastic knows the boundaries of the changesets it contains.

This might sound obvious, but Git users need to do this arithmetic themselves since the system doesn't track the origin of the branches.

To learn more about "diff math" [see this blogpost](https://www.plasticscm.com/download/help/diffmath) [<https://www.plasticscm.com/download/help/diffmath>].

MERGE AND CONFLICT RESOLUTION

Merging is the quintessential practice of version control. Master it and you will become a version control virtuoso.

This chapter contains everything you need to get a deep, full understanding of the craft of merging.

Pay attention to the details because you are going to walk the arcane paths of merge mastery.

Merge defined

Before diving into the details, I'm going to share the basics of what a merge is and what it is used for.

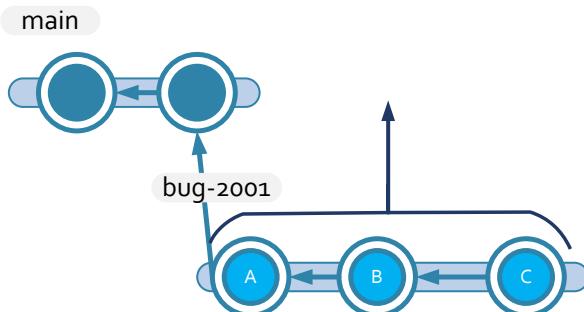
Born to merge

Branches are created to be merged. Only a small number of them stay independent, creating parallel dimensions of unmerged code.

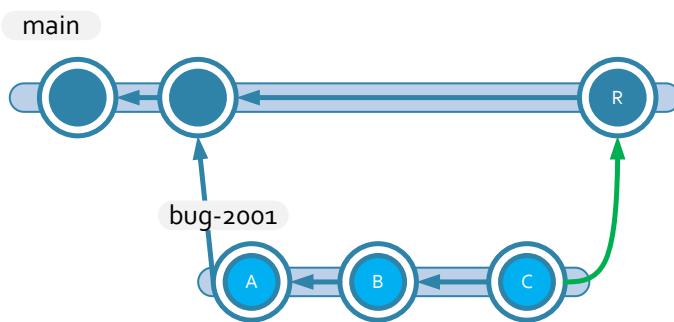
Merge from a branch

Merge is the operation to incorporate the changes made in a branch into a different one.

In the figure, to incorporate the changes A, B and C done in `bug-2001` into `main`, you need to merge `bug-2001` to `main`.

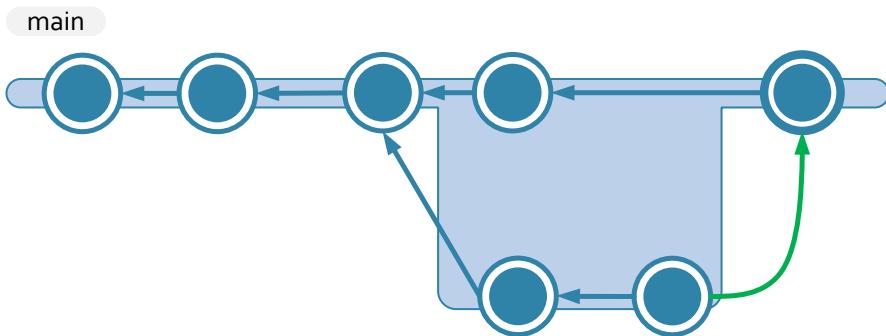


The result will be something as follows:



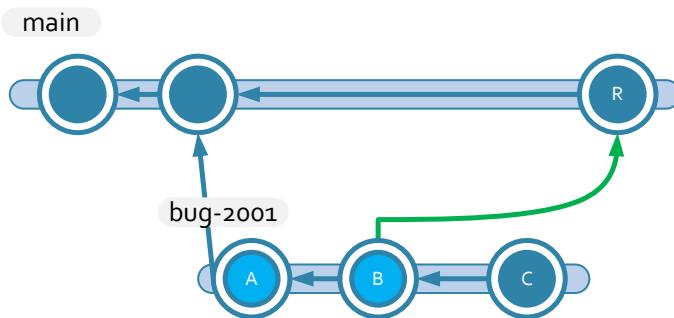
Where the resulting changeset **R** contains the changes made in **A+B+C** and *puts them in main*.

The merge from a branch is technically equivalent to the merge from a subbranch, since the same logic applies:



Merge from a changeset

Merge is also the operation to merge from a given changeset instead of taking a full branch.



If you merge from changeset **B** into **main**, you'll be getting the changes made in **B** and **A**, but skipping **C**.

Merging **bug-2001** is equivalent to merging **C** (the head of the branch).

There are ways to merge only the changes made in a changeset and not the ones in their parents. This is called cherry-pick and we'll be covering it in detail in this chapter.

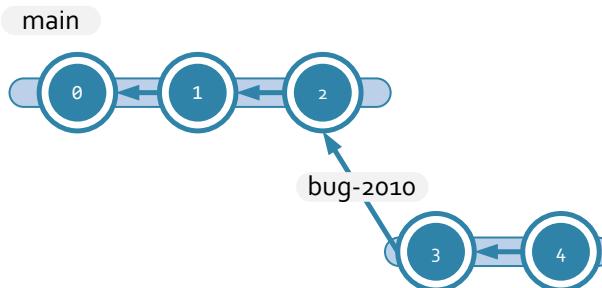
Merge contributors

Repositories are graphs of changesets

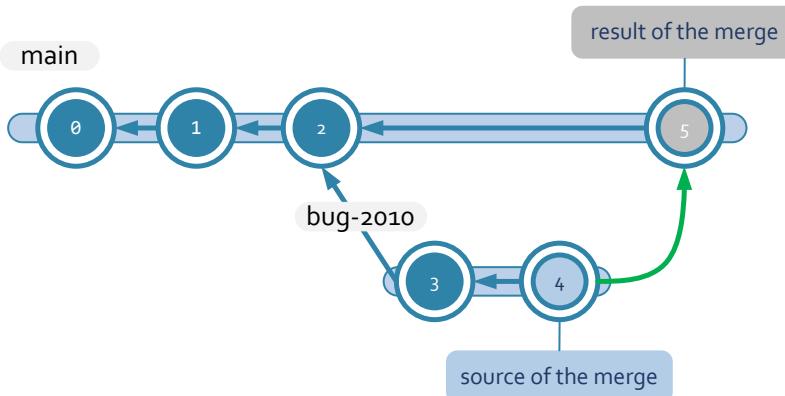
All through the book, you have seen a huge number of repository graphs. It is the same graphic Plastic renders in the Branch Explorer. It is a representation of the repository evolution through time. The history of the changesets and its branches, how they evolve, diverge and converge again.

Each time you checkin, you add a new node to the graph:

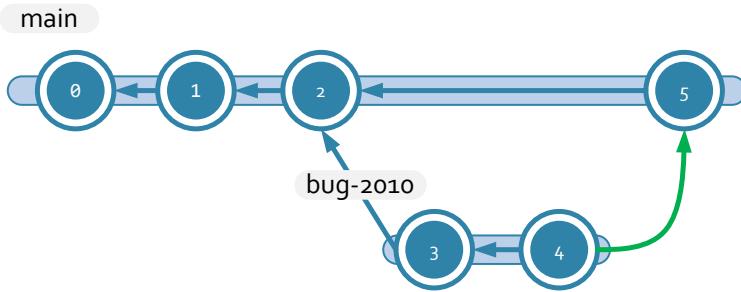
The arrow direction shows a *is parent of relationship*. This way 0 is the parent of 1. And 2 is the parent of 3. They might belong to the same branch or different ones, but what it is really important is child to parent links between them.



Finally, the merge links indicate the other type of relationship: source to result of a merge.



If now we remove the branch decorators, what is left is a typical graph with a single root (changeset 0) and then nodes that diverge in branches and converge again with merges.



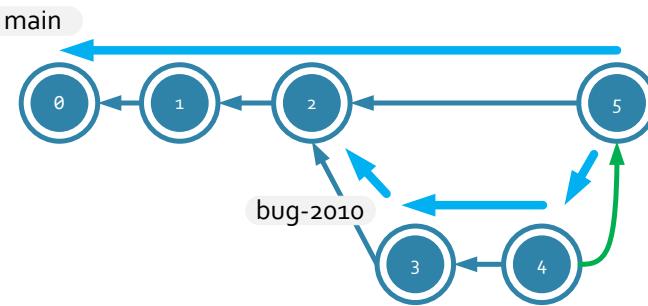
This graph is essential for us as users to understand the history of the repo, but it is also the key data structure that Plastic SCM uses to resolve merges.

Arrow direction

Now, let me explain why arrows in the Branch Explorer diagram are rendered the way they do.

We always say the Plastic changeset structure is a DAG: Directed Acyclic Graph. And I hear you saying: acyclic?? It is full of cycles thanks to the merge links! Well, visually yes, but internally for merge tracking calculation purposes, all links go backwards.

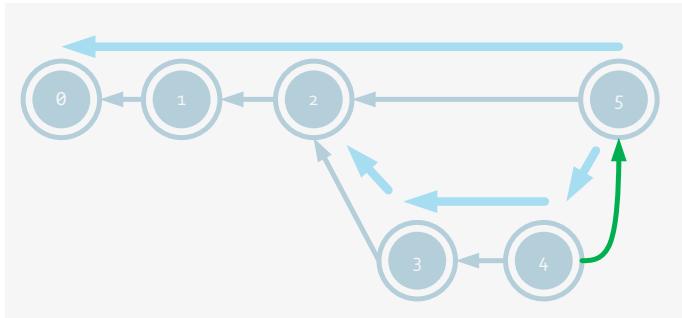
This is how Plastic walks the tree from **5** back to changeset **0**.



The first path is very clear. Simply back from **5** to **2** thanks to the parent link, then back in straight line to **0**.

The second path goes walking back the merge link from **5** to **4**, then back.

Why is it then that the merge link is displayed this way? If everything is heading back, why is the merge link the only one going in the wrong direction and actually creating a cycle?



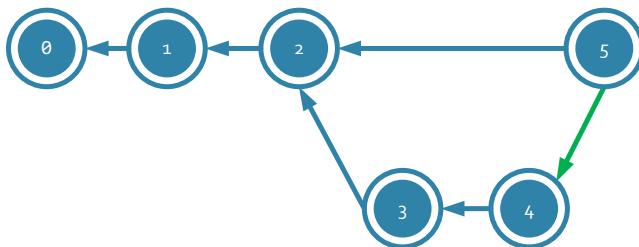
As I said above, merge links are always walked backwards. Always. From result to source. They are never walked in the direction they point to.

Why then do we keep them this way?

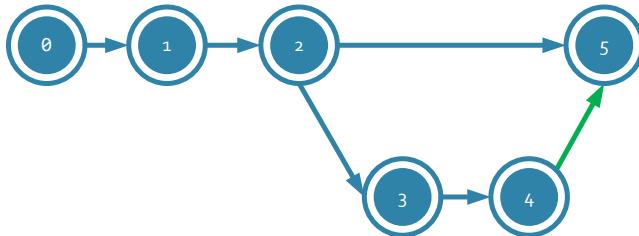
Well, when we talk about a merge we typically say: *merge changeset 4 to main*. You talk about a merge as *from* a branch or changeset *to* a destination branch or changeset. That's why we always considered it very natural to render merges this way.

Of course, there are other possible options, as the following diagram shows:

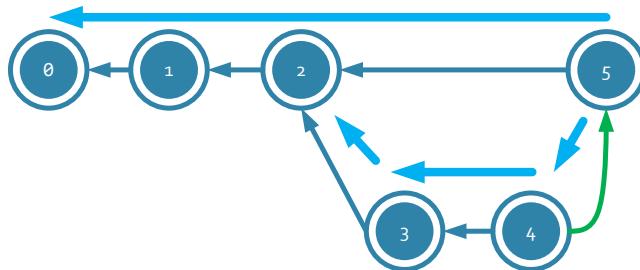
Option 1) Render merge links result to source. Opposite to current direction.



Option 2) Render changeset links as parent to child links.



Current) Changesets linked as we walk them during merge to find ancestors. Merge links as used in "merge 4 to 5".



Should we change how we render the Branch Explorer to option 1 or 2? It is not clear to me at this point, but I think it is really worthwhile to understand why everything is rendered the way it is.



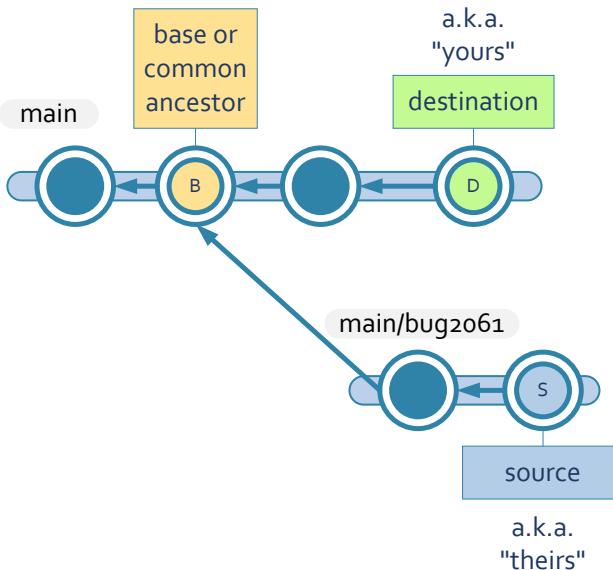
Customization

It is possible to render the Branch Explorer like in option 1 by setting a given flag in the configuration file. Check [this blogpost](https://www.plasticscm.com/download/help/arrowstory) [<https://www.plasticscm.com/download/help/arrowstory>].

Merge contributors: source, destination and base

Every merge has 3 contributors: the source, the destination and the base or common ancestor.

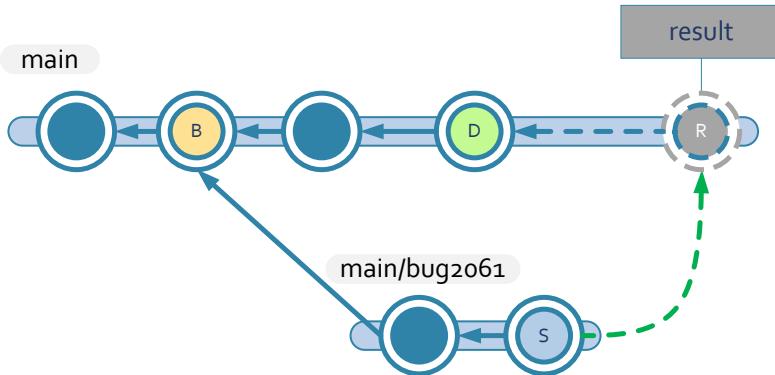
Check the following figure:



If you want to merge from `bug2061` to `main`, then the contributors are as follows:

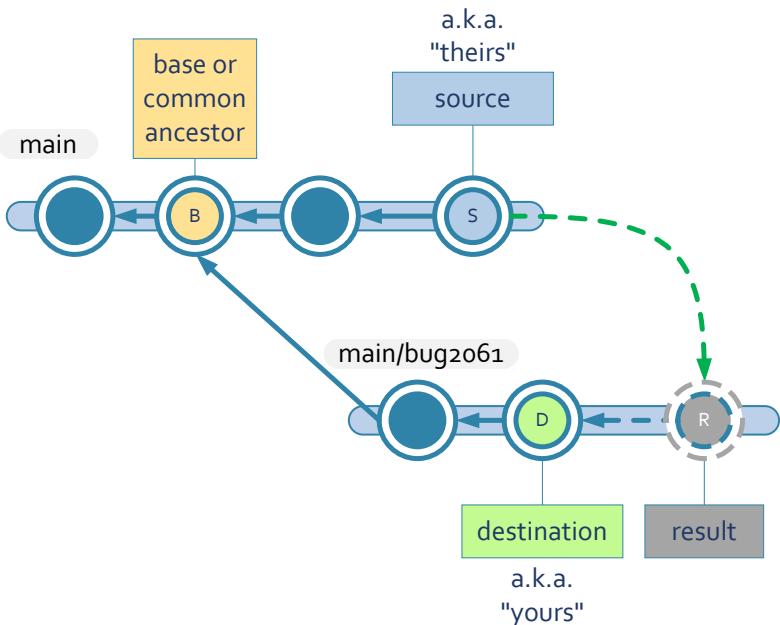
Source	Where you are merging from. In this case, the changeset marked as S. It is also known as "theirs".
Destination	Where you want to put the result of the merge. Marked as D in the diagram. It is also known as "yours" since this is typically where your workspace is when you do a merge.
Base	The <i>common ancestor</i> of the other two contributors. If you walk the graph back from D and S, B is the first node they have in common.

Once the merge is started, the resulting graph will look as follows:



Please note how I used dashed lines for the result and the merge link pointing to it, and the line connecting the result with its parent, the destination. This is how we draw in-progress merges. Once the changeset that is the result of the merge is checkedin, the lines will be uniform and not dashed.

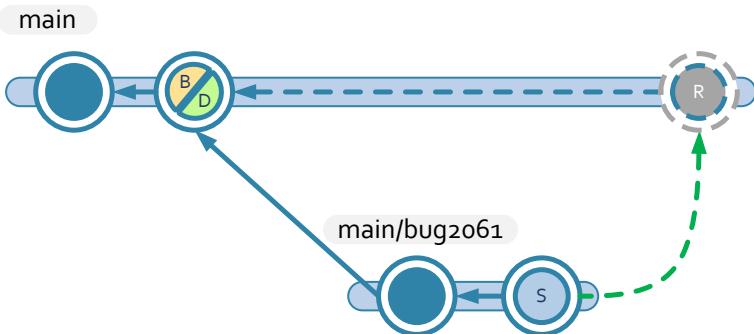
What if you decide to merge from `main` down to `main/bug2061` instead? If that is the case, then the contributors will be slightly different:



Basically, the base would remain intact, but source and destination would be switched. Result is always a child of the destination, so it would now belong to **main/bug2061** instead.

The cases above are considered the canonical ones where the source and base are separated by at least one changeset.

Let's see a new scenario:



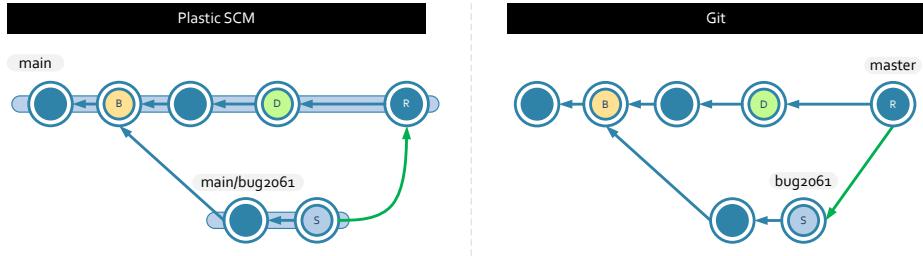
In this case, you want to merge from **main/bug2061** into **main**, but there weren't new changes in **main** after **bug2061** branched off. Then, what happens is that the same changeset is simultaneously base and destination. This is a very typical case when you create a couple of branches and then merge the first one back to **main**. There is no possible conflict because the actual topology of the merge says no concurrent changes exist.

Plastic always creates a changeset as result of the merge

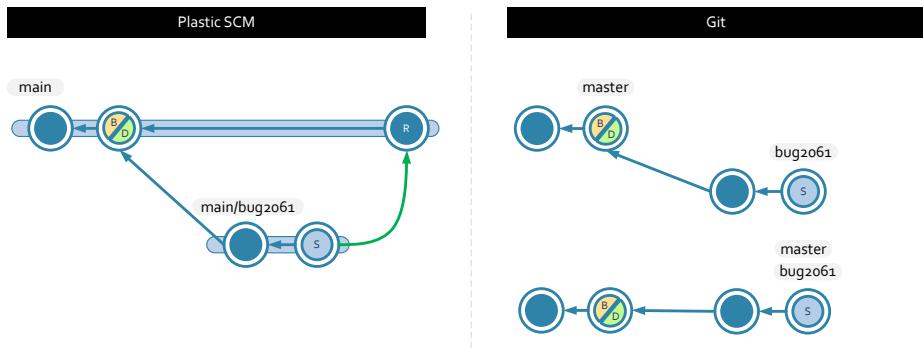
It is very important to note that Plastic always creates a changeset as result of a merge.

I highlight this because this is radically different than what happens in Git, and since many of you were exposed to Git prior to your jump to Plastic, it is very worthwhile understanding the difference.

The first scenario is a case where the destination and the base are not the same changeset/commit. Topologically, this is a case where conflicts might potentially happen. As you see in the figure, Plastic and Git behave in the same way. Both create a result changeset that reflects what actually happened.



The second scenario happens when base and destination are the same changeset/commit. In other words, if no possible conflicts can happen due to the repo topology:



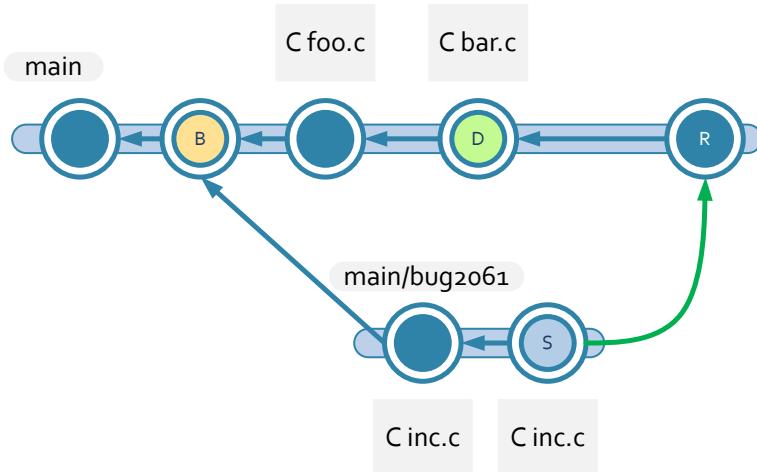
Here Git performs a *fast-forward*. There are no possible conflicts so all it does is to move the master to the same commit where `bug2061` is. It is quite efficient because new commits are not created. The drawback is that the history of the branch is lost, and you can't figure out anymore which commits were created as part of `bug2061` and which ones as part of `main`.

In Plastic, we always create a result changeset. This is because a changeset can't be in more than one branch at the same time, and also because it is very important for us to properly [preserve the branch history](#).

Graphs with potential merge conflicts

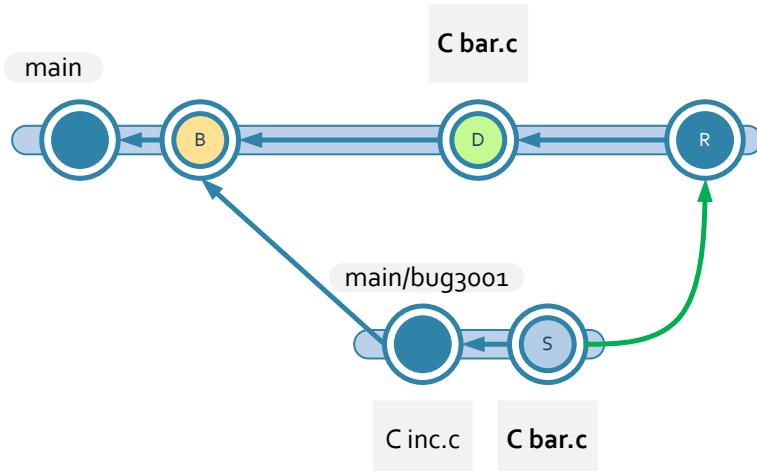
As you saw in the previous section, there are cases where there are no possible conflicts. If the base and the destination are the same changeset, then all Plastic has to do is to add the right files to the resulting changeset since no possible conflicts can happen.

The other scenario is when the shape of the graph tells us that there are potential conflicts. I say potentially because more often than not, even when the graph has concurrent changes, there might not be possible conflicts. Let me show you why:



In the figure, you see how in `main/bug2061` only `inc.c` was modified while in `main` after `bug2061` was branched, only `foo.c` was changed. This means while, at first sight, there might be potential conflicts in the graph because there are concurrent changes, in reality no files were changed concurrently.

The same graph topology can lead to conflicts when concurrent changes on the same file happen, as the branch `bug3001` shows in the next figure:



Here `bar.c` was changed both in source and destination which provokes a file conflict. The conflict can still be automatically solved as we'll see in 3-way merge, but there will be cases where manual user intervention will be necessary.

2-way vs. 3-way merge

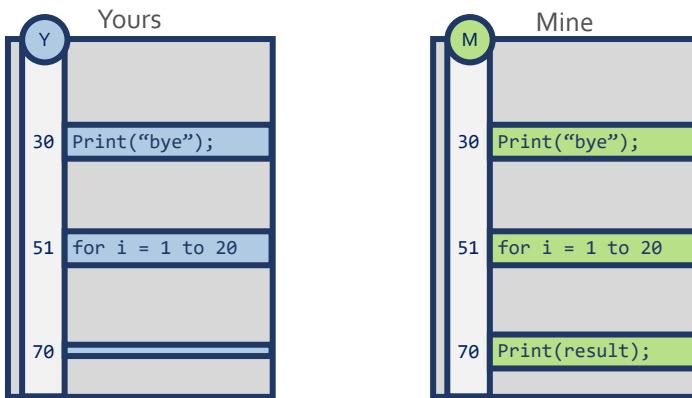
Before we jump into how merge tracking actually works and why it is so useful, I think it is a good idea to cover the difference between 2-way and 3-way merge.

Merge tracking, as you will see in a moment, is all about finding a "common ancestor". We already saw how every merge is about finding 3 elements: the base, the source and the destination. Now, why is this base thing so important?

The answer lies in the difference between 2 and 3-way merge.

2-way merge: life before common ancestors

Look at the following case:



Did you delete line 70, or did I add it?

There is no way to know looking at the two files. Only our knowledge of the actual file would tell.

This is a 2-way merge; you merge 2 files by simply comparing one to the other.

As you can see, there is no way to merge files with 2-way merge without manual intervention. Conflict resolution can't be automated at all.

This is how old version control systems (namely SVN and CVS among a few others) used to work, and that's the primary reason why an entire generation of programmers all around the world developed "merge phobia". Can you imagine having to merge 300 files when all merges need mandatory user intervention even if changes were trivial?

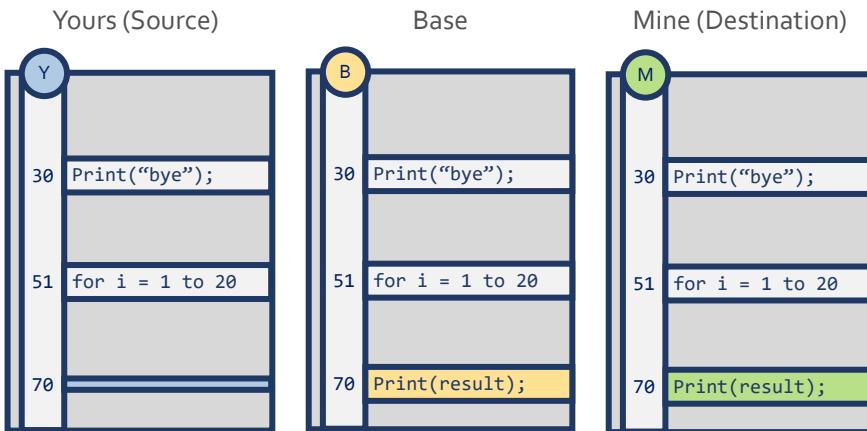
That's how a world without common ancestors would look like 😊

3-way merge

Let's forget about version control for a second and let's simply focus on how to manually solve a conflict like the one in the previous example. How can anyone solve that without having previous knowledge about the project?

Simply: he would just need to check what the file looked like before the two contributor made changes.

This "how the file was" is what we call the base. If we use it to solve the conflict, then the scenario will be as follows:

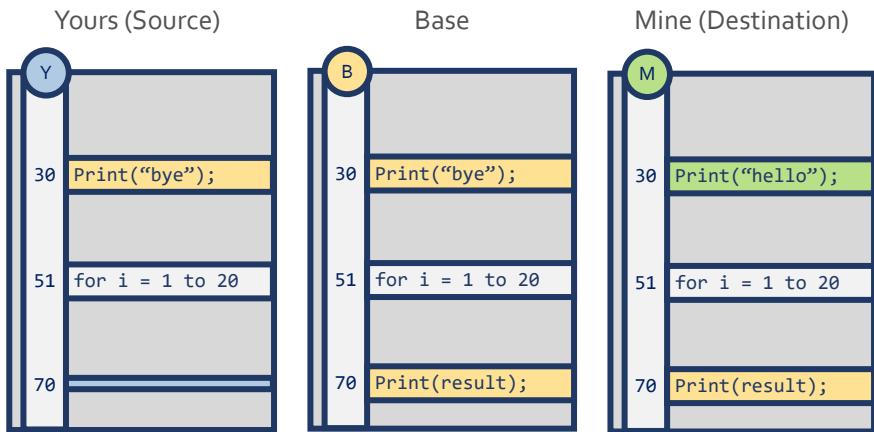


As you can see, the conflicting line 70 already had the content `Print(result);`; originally. So, it is clear that it was deleted on source (yours) and untouched on destination (mine). The conflict can be automatically resolved now; all we need to do is to keep the deleted line.

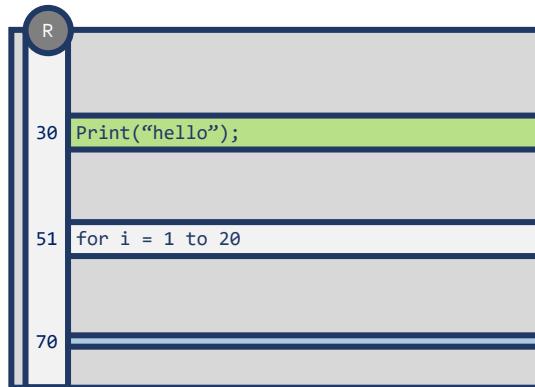
This is the magic of 3-way merge. Most of the actual file conflicts will be automatically resolved using very simple logic: if you modified one line and I didn't touch it, then yours is the one to keep.

The good thing here is that 3-way merge can solve many conflicts in a single file. Let's go to a new example: this time both lines 30 and 70 enter the scenario. First, one user modified line 30 while at the same time the other deleted line 70. Still, a 3-way merge tool can merge the file without manual intervention. The figure shows the base and the two contributors and then the result.

As you can see, applying very simple logic, it is possible to conclude that we must keep line 30 from destination (marked in green in the result to highlight the chosen contributor). Line 70 must be kept from source like in the previous example, and the result marks the deleted line in blue to reflect the contributor it is coming from.



Result - automatic



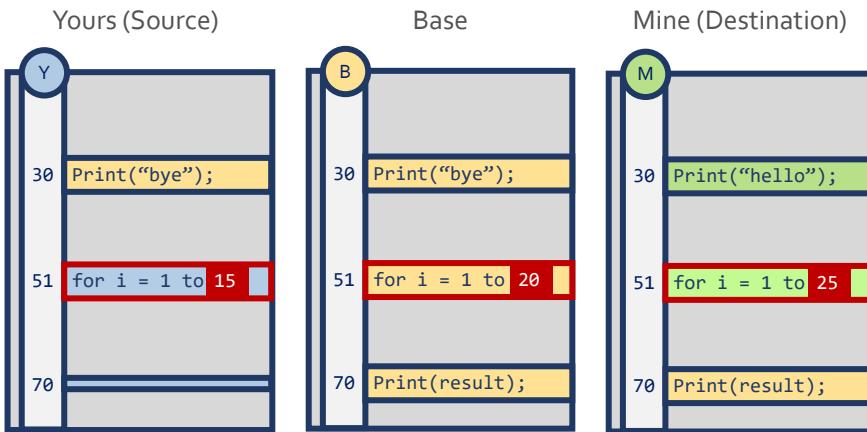
Now imagine you have to merge from a branch where 300 files were changed while the same 300 files were also modified in the destination branch. Thanks to 3-way merge, most likely the merge won't even require manual intervention because the very simple logic behind the text merge algorithm will be able to solve all conflicts automatically.

The actual logic to automatically solve conflicts can be summarized as follows:

Source (yours)	Base (common ancestor)	Destination (mine)	Result
A	A	B	B
B	A	A	B
B	A	B	B
B	A	C	Manual conflict

Now, what if two developers modify the same line divergently in two different ways? Check the following figure where the two developers modified the for loop differently. It was from 1 to 20 originally, but then one decided to set it as 1 to 15 while the other changed to 1 to 25.

In this situation, the merge tool doesn't know what to do and will simply ask for human intervention. This is what we call a manual conflict.



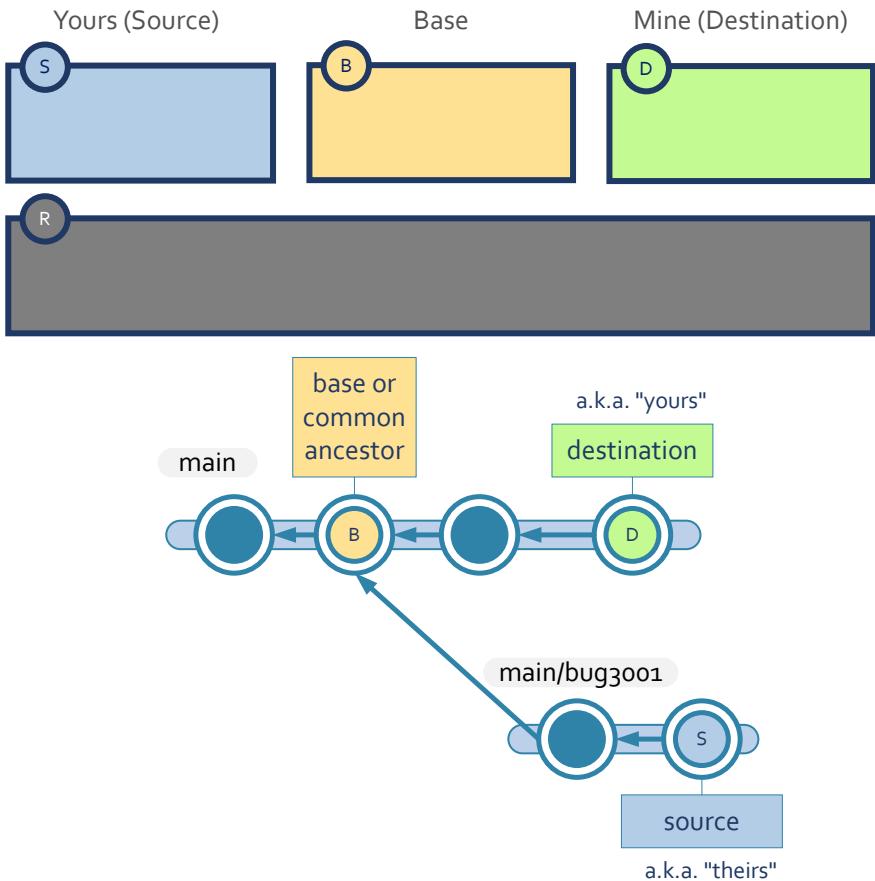
As you can see, the key is to have a base or common ancestor to be able to run 3-way merges. Otherwise, all merges would be manual.

All the links and arrows in every single diagram in the book have a key purpose: provide the merge algorithm with the information it needs to find the common ancestor between any two nodes in the graph. This is the way to find the right base for every single file involved in a merge.

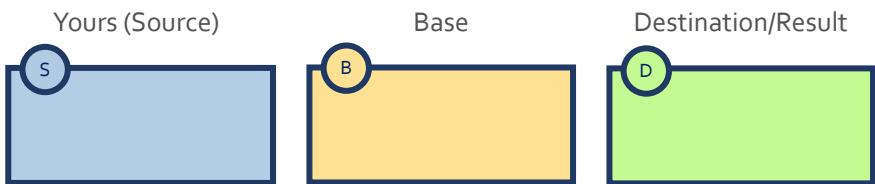
Layout of 3-way merge tools

Are you familiar with any of these tools? Kdiff3, p4merge, BeyondCompare, WinMerge? They are all merge tools. Plastic comes with its own 3-way merge tool, called Xmerge and we think it is much better than all the others. But my point here is not to sell you the merits of our tool, but simply explain what they all have in common.

As you saw in the examples above, when you do a 3-way merge you need to handle the source (the changes you are merging from), the destination (your copy) and the base (how the file was prior to any of the changes). The drawings in the previous sections showed a very typical layout. It is in fact the one we use:



An alternative one, used by many merge tools, unified the result/destination as follows:



Personally, I prefer the first option (I mean, that's the one we use after all), but what it is important for you is to keep in mind that there are two main alternatives. Once you know that, no merge tool will be difficult for you. They might have more or less buttons, a simpler or more cluttered UI, but once you know the key elements in any tool, it is just a matter of just finding what you are looking for.

Merge tracking

All the arrows and nodes I drew so far in the book have a dual purpose: one is to explain you, the developer, how your repo evolved. The other is to resolve merges.

This second part is what the Plastic core really cares about.

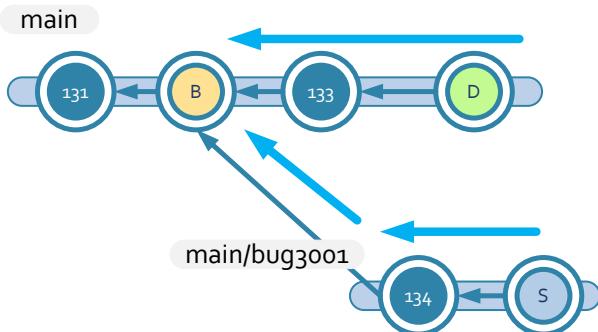
Whenever you launch a merge, the first thing Plastic has to do is to identify the three contributors of the merge. Two of them are obvious because you specify them: the source and the destination.

Merge tracking is all about calculating the correct common ancestor or base.

Calculating the common ancestor

To find the common ancestor of two given changesets, the Plastic server takes the changeset graph (basically the same structure you see in the Branch Explorer) and walks back the tree until the paths converge.

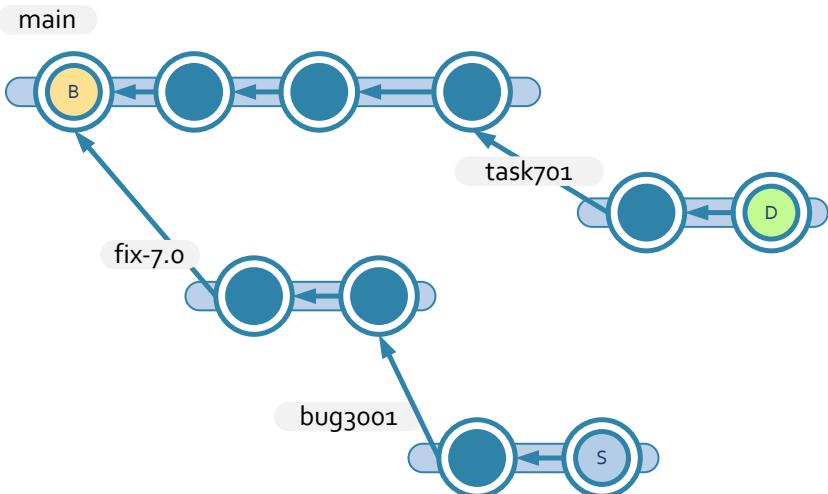
As usual, it is much easier with an example:



The algorithm starts two walks from D and S and their paths collide first on B, which is the base or common ancestor.

In fact, the base for the merge is the *nearest common ancestor*, because in the example the changeset 131 is also a valid common ancestor, but it is not the closest to the two contributors.

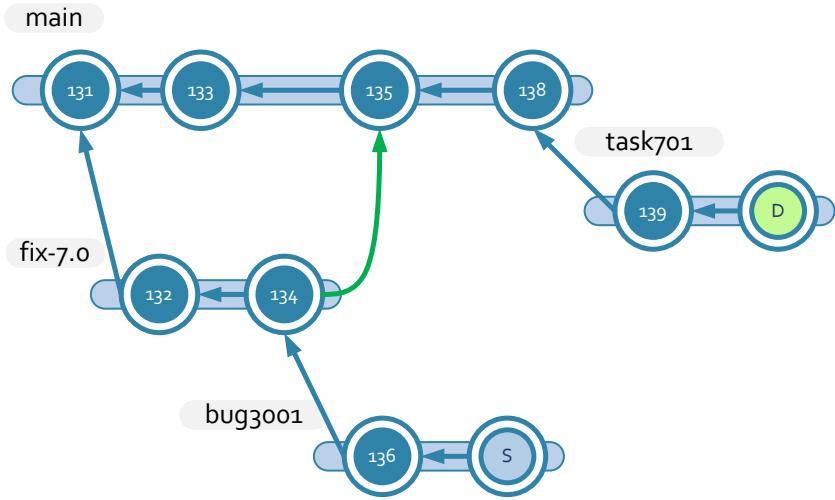
Of course, life is not always that simple and finding the common ancestor is not that obvious as in the previous example. Check the following figure:



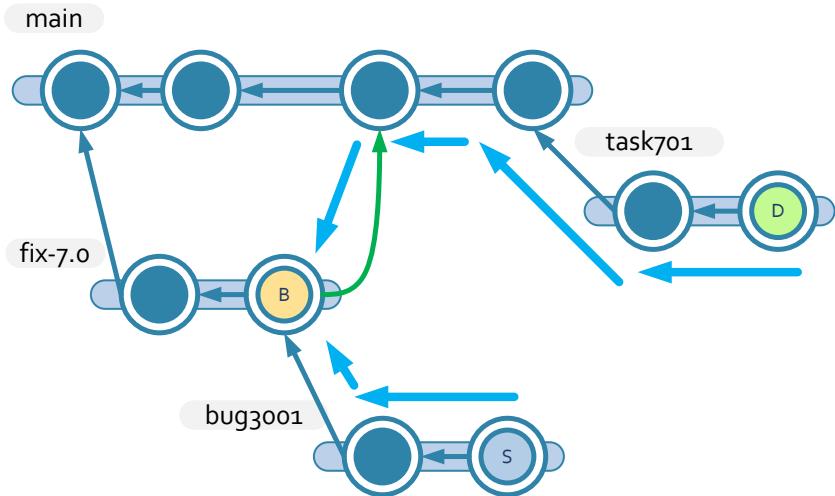
Note that the case is not super difficult either, but as you can see you have to walk through several branches before finding the changeset marked as B. It is a pretty obvious case though, exactly like the first one but just with a longer tree walk to reach the base.

What makes merge tracking slightly more complicated is that every new merge is affected by the previous merge history (merge history is equivalent to merge tracking to me, so you can also use it interchangeably).

Try to find the common ancestor in the next figure: (don't go to the solution until you make a guess 😊)



I want to merge branch `bug3001` into `task701`, so Plastic will walk back from S and D and will consider the merge arrows as traversable paths. So, the result will be as follows:



Yes, changeset `134` is the actual common ancestor thanks to the merge link. Otherwise, it would have been `131`.

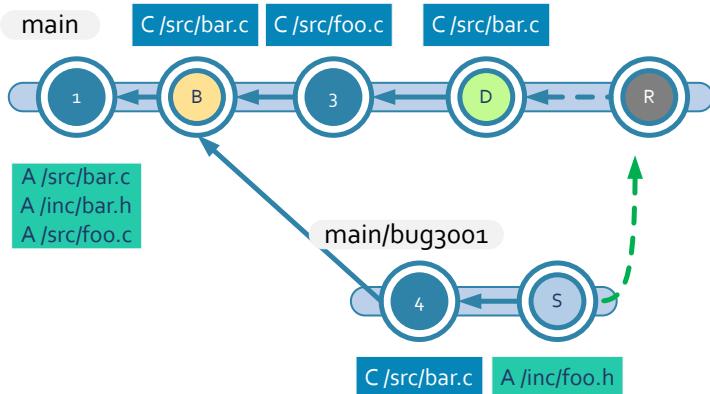
All these cases I described so far are pretty simple, and in fact this is how most typical merges will be most of the times. There will be situations, though, when more than one common ancestor will be found

at the same distance. And then recursive merges will enter the scene. But, before going any deeper into merge tracking, let's discover what happens once the right common ancestor is found.

Merging trees

We arrived at the core of the merge process! You are about to become a master in the internals of merge, and gain a deeper understanding of how Plastic actually merges branches. Armed with this understanding, you will no longer have trouble understanding even the more complex merges.

Of course, we are going to start with a branch diagram:

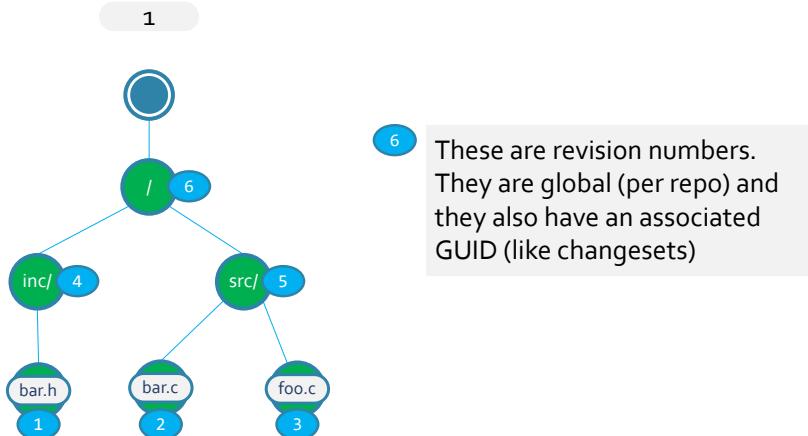


We want to merge `bug3001` into `main`. The common ancestor is already located and the diagram is decorated with the actual changes.

To calculate the merge, Plastic doesn't go and load all the individual diffs made in 3 and 4 and then destination and source. It doesn't accumulate the changes to find the solution. (This is not a bad alternative and some systems like Darcs do something similar, but we do it differently).

What Plastic does is to load the tree at base, source and destination and then calculate the diffs between source and base, then destination and base and work based on that.

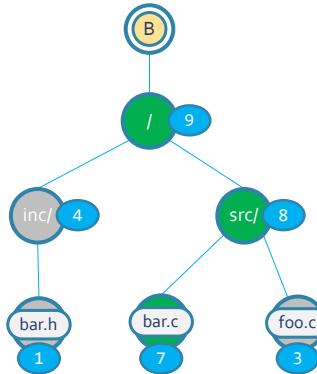
Let's go step-by-step. First, I'm going to create the tree for changeset 1:



The numbers are the revision ids. Every object in Plastic has an object id, which is an int (4 bytes). It is also identified by a GUID that makes it unique even across replicated repos. I'm going to use the revid (revision id) for short but the example would also work with GUIDs.

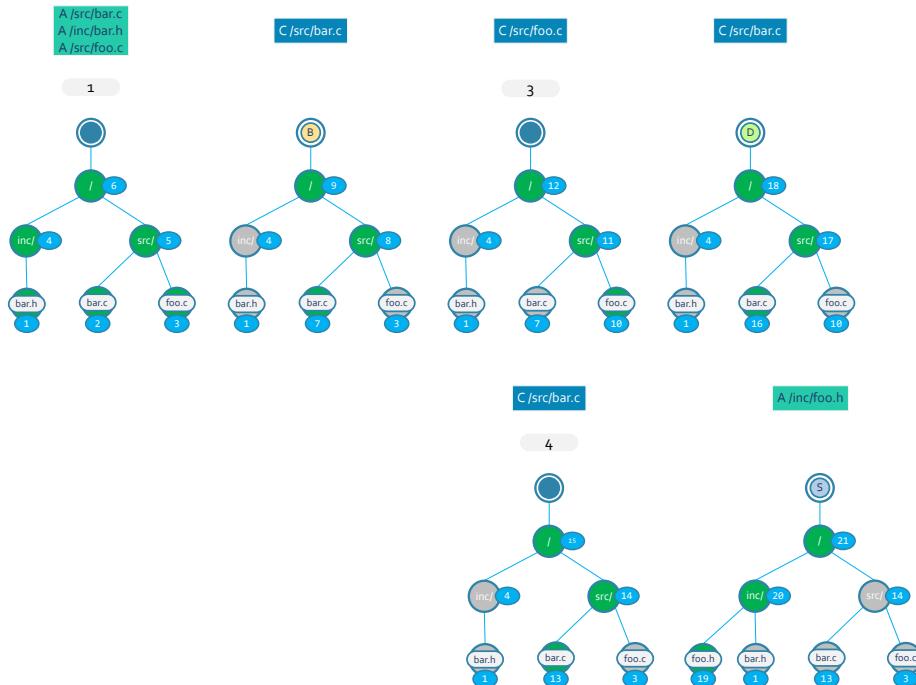
It is also important to note that every file or directory also has an item identifier (item id). This means `bar.h` might have many different revisions, but they all belong to a given item id.

Now, let's go to changeset number **2**, the base, where we only changed `/src/bar.c`. This is what its tree looks like:



I marked in green the nodes that were modified from the previous changeset **1**. As you can see, there is a new revision for `bar.c`, then for its containing directory `src/` and then the root `/`. The rest of the tree remains unchanged.

For the sake of clarity, I drew all the trees in the example:

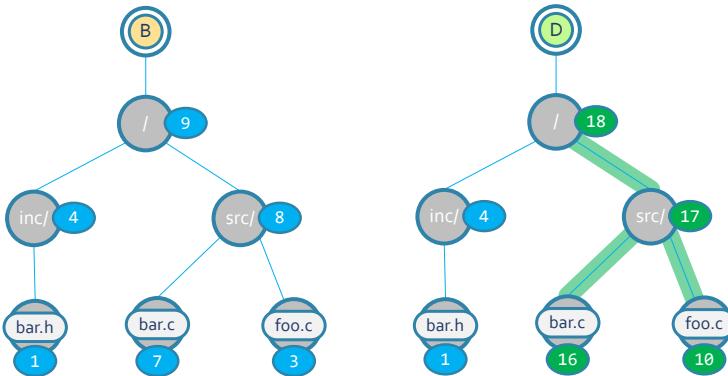


I think this graphic gives you a pretty good explanation of how Plastic actually works. Of course, it is very good to understand merge, which is our goal now.

As I said, to figure out the actual changes since the base Plastic doesn't load all the trees, it just loads the base, source and destination.

Once the trees are loaded, Plastic first calculates the diff between the base and destination trees. It will be something as follows:

Diff base and destination



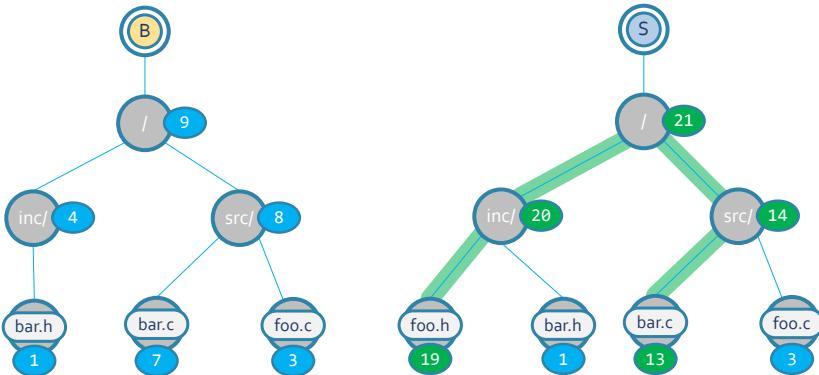
The merge code basically walks the two trees as follows:

- Roots are different. `revid 9 != revid 18`. So, continue walking.
- Take `inc/`. `revid 4 == revid 4` in both trees. Diff can prune the `inc/` subtree.
- `src/ revid 8 != revid 17`. Continue walking.
- Then, it finds that both `bar.c` and `foo.c` differ, so both will be marked as different.

Diff(base, destination) = changed `/src/bar.c` and `/src/foo.c`

Now, the differences between base and source:

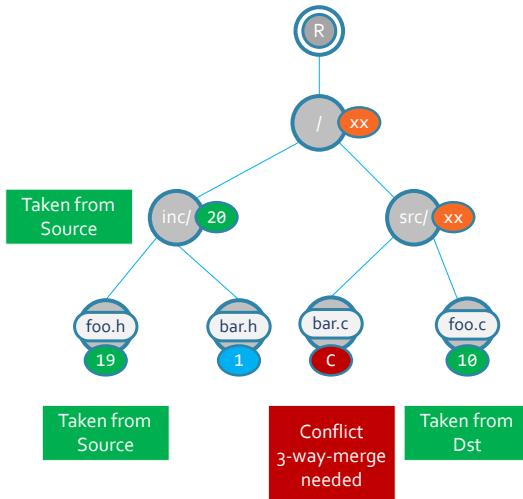
Diff base and source



Here:

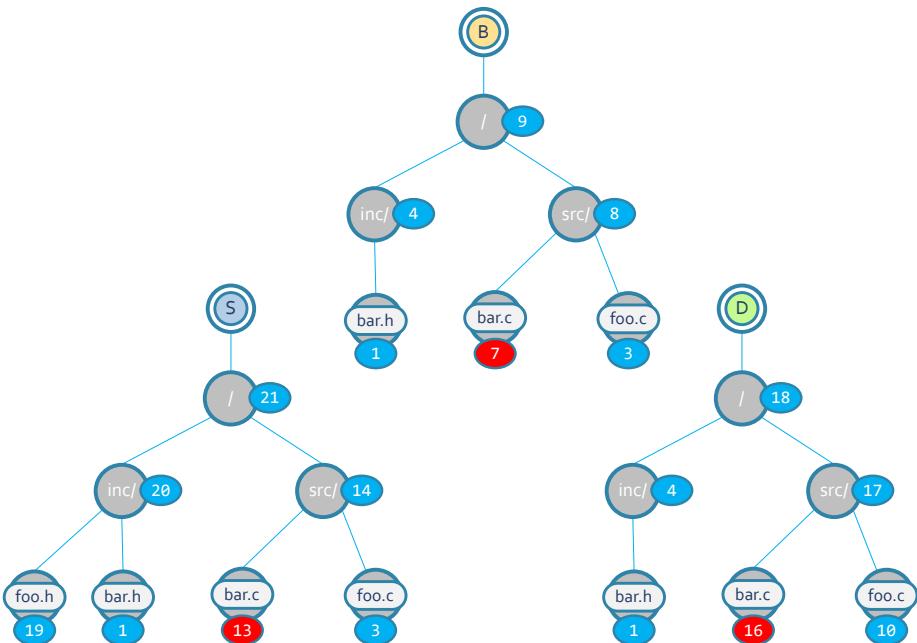
Diff(base, source) = added `/inc/foo.c` and changed `/src/bar.c`

Now, the algorithm takes the two pairs of differences to find conflicts. It can mount a result tree like the following:



As you can see, it can already mount `/inc/foo.h` in the tree, and it knows it has to load `/src/foo.c` revid **10**. But, there is a conflict in `bar.c` because concurrent changes happened on both contributors.

Let's now put the three trees together:



To solve the conflict in `bar.c`, we'll need to run a 3-way merge loading [revid 7](#) as base, [13](#) as src and [16](#) as dst.

And this is how everything comes together! Finding the common ancestor to actually locate what changed in the two contributors and then locating the common ancestors of the files in conflict. ☺

And I think I never drew so many graphics before in my life ☺

If you went through all this and really got it, now you can say you are an expert in merge. Congratulations!

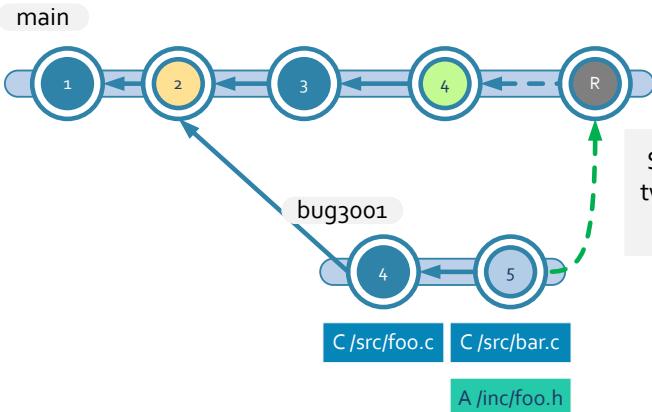
Changeset-based merge tracking

Merge tracking happens at the changeset level, not the file level.

When Plastic tries to find the common ancestor of a merge, it does it on a tree-wide level.

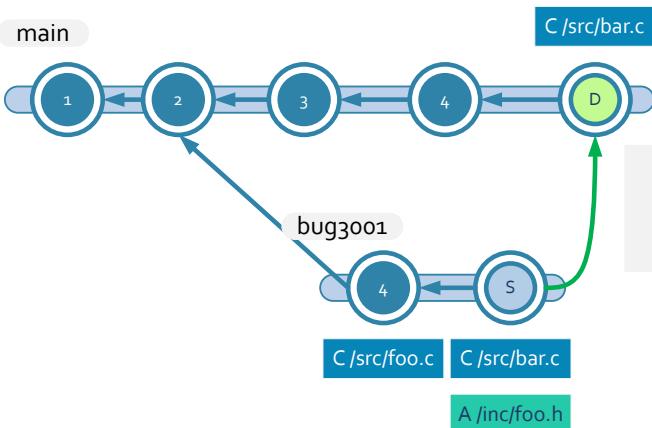
I'm going to explain what this exactly means and the implications it has.

Consider the following scenario:



You only want to merge `/src/bar.c` to `main`, but skip the others because you are not interested in them for whatever reason. Plastic can't do that, but suppose for a moment that you manage to make it happen.

Now, you'll have a resulting changeset where only `bar.c` was merged from `bug3001` and a merge link set, as follows:



And then you want to finally get `foo.c` and `foo.h` into `main`. You try to repeat the merge but the merge link between source and destination clearly says no merge is possible because there is a direct path between them (source is a parent of destination so no merge can happen).

Do you see the problem now? That's why when you merge a branch or changeset, you merge all or nothing, because otherwise we'd render the merge links useless.



There is one alternative we are considering to give more flexibility to users which is allowing the first merge to happen but without setting the merge link. Maybe we can set some sort of informative link so you visually know something happened (although the algorithm will ignore it to find ancestors). Then, the next time you want to merge, you'll be able to complete the operation.

The alternative to this is per-item (per file, per directory) merge tracking. Plastic and Git (and I'd say any modern version control) uses per-changeset, but SVN, Perforce, TFS, and the now archaic Clearcase, use per-item merge tracking. In fact, we used per-item prior to version 4.0.

What are the cons of per-item?

- Speed. When you have a tree with, let's say, 300k files, you have 300k different merge trees, one per file and directory. You can optimize the calculation as much as you can, but believe me, it will never be as quick as simply having to traverse one single merge tracking tree.
- Conflict resolution. With changeset-oriented merge tracking, you merge trees. Fixed trees associate to changesets. This enables solving a number of directory conflicts that were simply impossible by the per-item merge tracking. We're going to navigate directory conflicts soon and you'll find out why they are useful to have in your toolbelt.

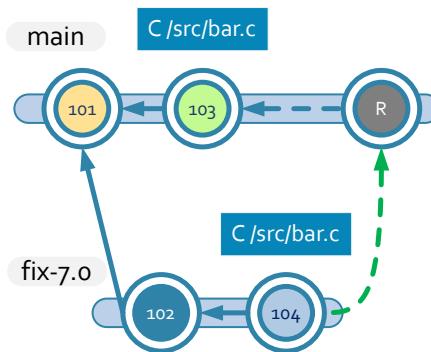
Little story

Back in 2011, Plastic was adopted by a large team of 1000 developers who worked on a single huge code base. The main reason to abandon Clearcase and go to Plastic was that large merges decreased in time from several hours to just a few seconds. Per-changeset merge tracking was the reason for the increase in speed.

Why merge tracking matters

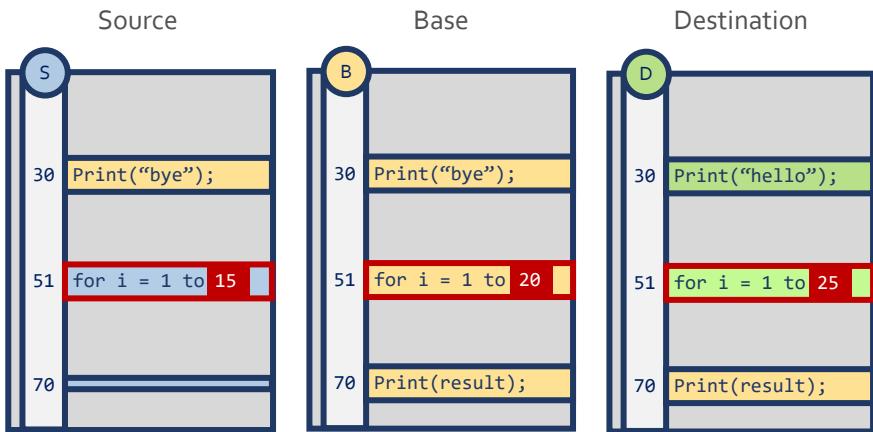
Merge tracking can save tons of manual work by automating merges that otherwise would require manual intervention. We already saw it with some simple 3-way merges. Now, I'm going to walk you through a very interesting case where you'll discover why having proper merge tracking makes a huge difference.

Let's start with this scenario:

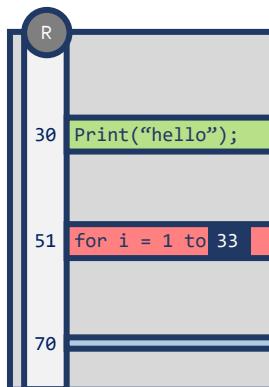


We want to merge **fix-7.0** back to **main** and there is a conflict on **/src/bar.c**.

Let's take a look at the actual files and solve the conflict:



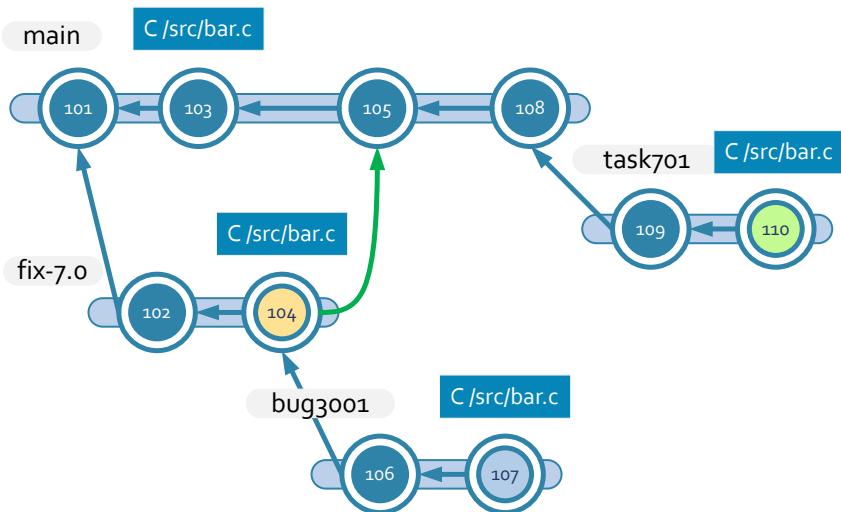
Result – solved manually



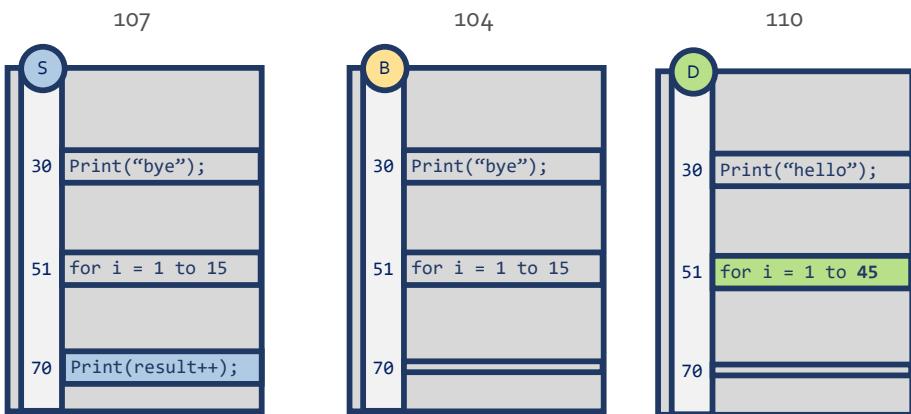
Look at the figure carefully:

- Line 30 was only modified on destination, so the Print("hello"); is kept.
- Line 70 was deleted on source, so the deletion will be kept too.
- The problem is in line 51 because it was changed divergently. Manual intervention is required and then we decide to keep it as for i=1 to 33. We didn't keep any of the contributors, we just decided to modify the result manually.

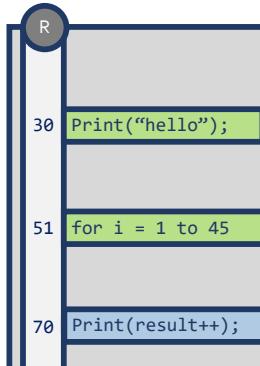
Then, we continue working on our project making a few more changes as follows:



And now, we want to merge `bug3001` into `task701`. The common ancestor will be changeset `104` (the old source this time). Let's zoom into the file contents to solve the possible conflicts:



Result – solved automatically

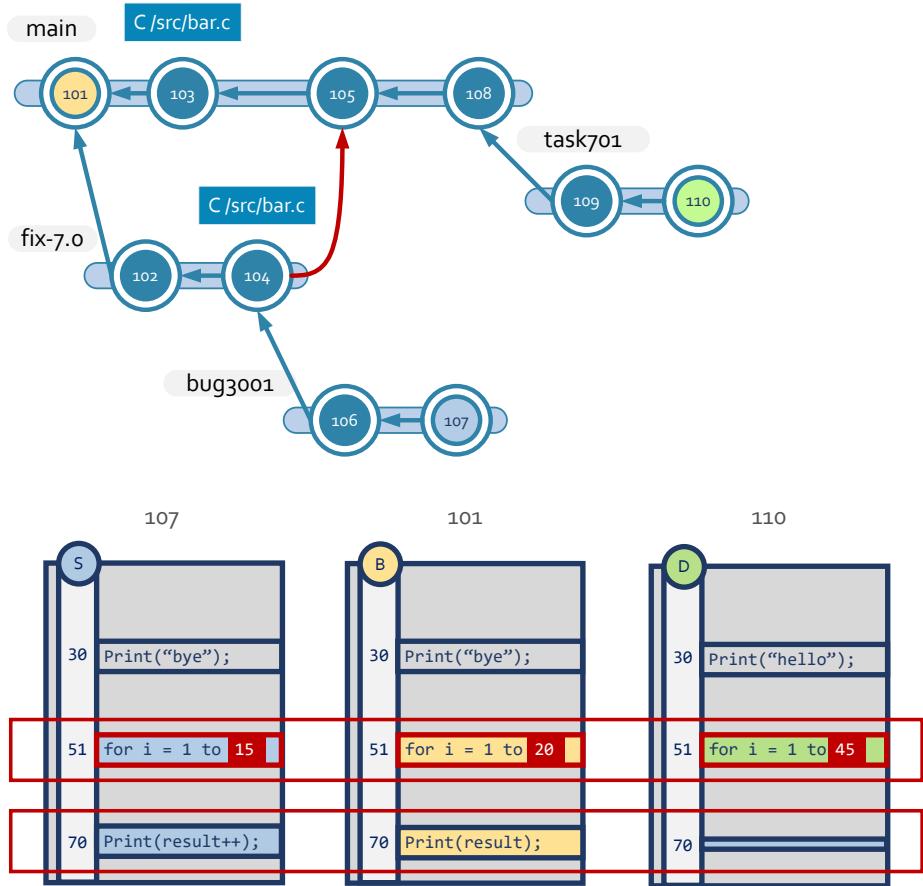


What actually happened is that in `task701` that line 51 was modified so that it is now 1 to 45. Meanwhile, in branch `bug3001`, line 70 was added with content `Print(result++);`.

The result of the merge can be solved automatically by a 3-way merge tool.

Well, you may say, this is pretty normal, the 2 branches don't touch the same line, so why should it be manual?

Let's take a look at how the merge would be if we ignore the merge link between 104 and 105:



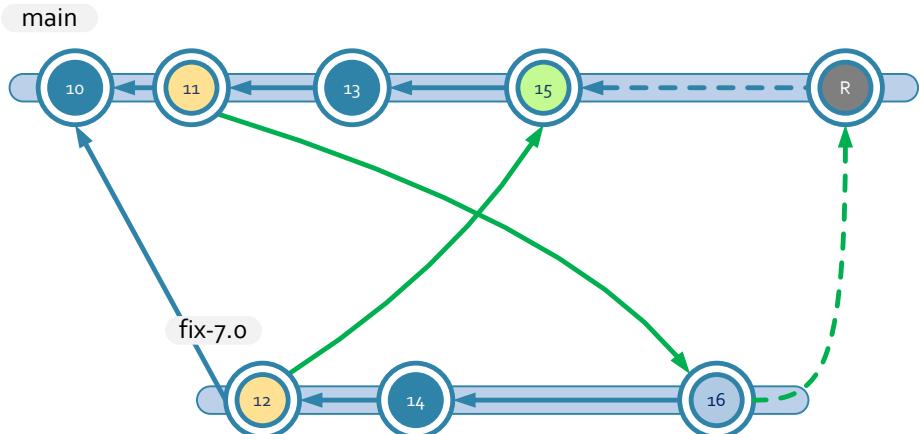
Not only the merge is not automatic anymore, but you have two manual conflicts instead of one!!

Take your time to look closely at the figure and really understand why there are manual conflicts now. It will really pay off.

Recursive merge - intro

In all the examples, in this chapter, we always had two contributors (source and destination) and one common ancestor. All the trouble was simply finding the ancestor, but the only difficulty was having to walk the tree a little bit further. Of course, the ancestor could be harder to find in a forest of branches, but for an algorithm, it wouldn't be that hard.

Now, what if there is more than one ancestor found at the same distance? Let's see an example:



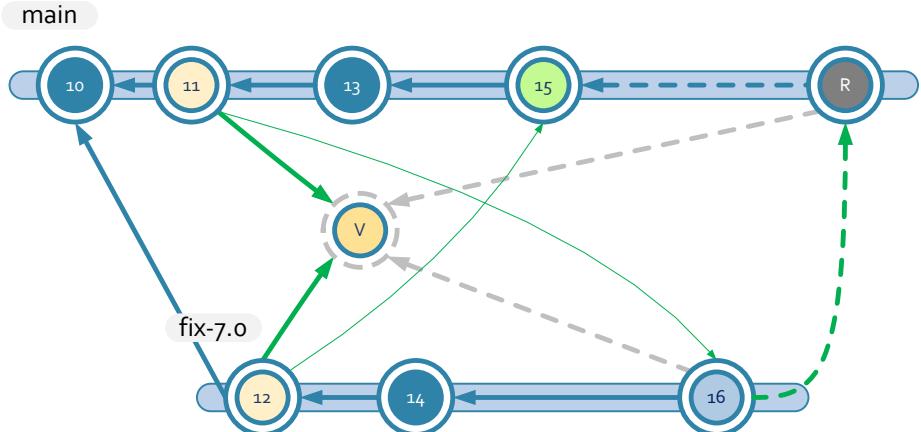
We call this a crisscross merge scenario. The case itself is a little bit forced because there is apparently not a good reason to merge from 11 to 16 when 15 already exists. But, as you can imagine, this could really happen in a distributed scenario where the merge 11 to 16 happen on a repo where 11 is the head of `main`, while 12-15 happens on a different one where only 12 was pushed at the time.

Anyway, the scenario in the figure is the shortest one driving to a merge with multiple ancestors, but the same can happen with several changesets and branches between the "crossed" merges, as we'll see later.

As you can see in the figure, both changesets 11 and 12 are valid nearest common ancestors because they are both at the same distance from 16 and 15. Which one should we choose?

The answer is not actually choosing one because a random choice will lead to incorrect merges, but running a recursive merge where a new unique virtual common ancestor is calculated.

The figure shows the new virtual ancestor that is created by Plastic by merging changesets 11 and 12 and then its resulting tree is used as ancestor of 15 and 16 to resolve the merge.



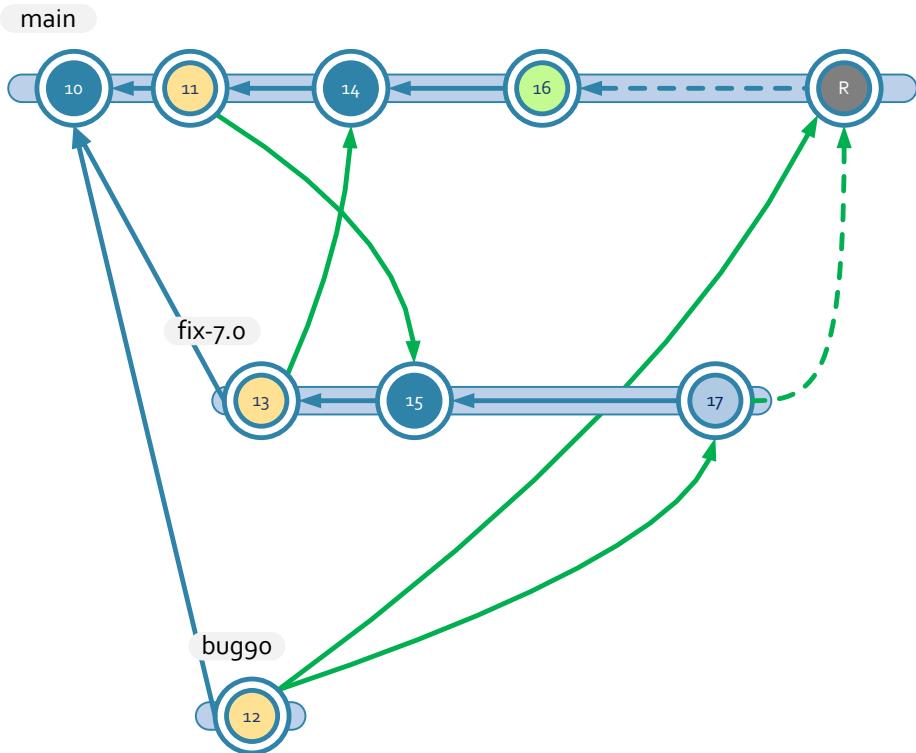
Plastic does all this operation under the hood when it finds multiple ancestors, so normally you are not even aware of what is going on and the merge simply happens correctly.

There is some effect you can notice though. Suppose a file `foo.c` is modified both by [11](#) and [12](#), so it needs to be merged and there are manual conflicts. Plastic will ask you to solve those conflicts to use the resulting file as basis for the next merge. Then, if the file was again modified in [15](#) and [16](#), you could have to solve a manual conflict again. The effect is that you'll see the merge tool show up twice for the same file. Now you understand why it can happen 😊

Recursive merge – more than 2 ancestors

Once the basics of recursive merge are clear, let's jump into a more complex case, where more than two ancestors are found at the same distance.

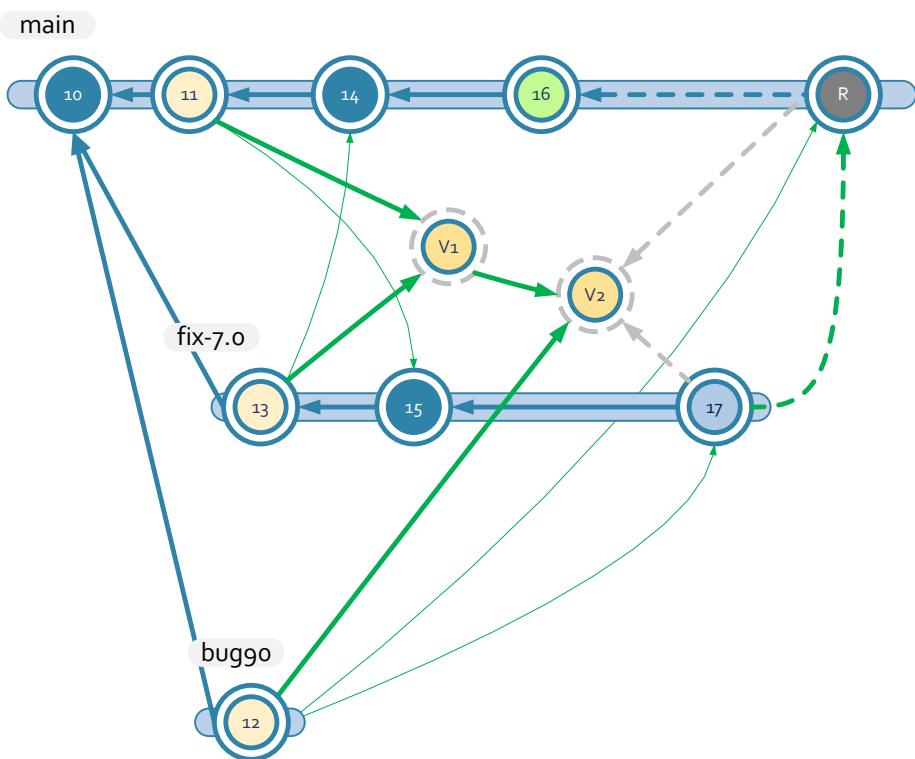
Take a look at the following figure:



As you see, now we have three ancestors at the same distance: [11](#), [12](#) and [13](#). How does Plastic solve this issue?

It will chain virtual ancestors merging them in pairs recursively (that's where the name actually comes from) until only one is left. This final virtual ancestor is then used to solve the original merge.

The next figure explains how the ancestors [V1](#) and [V2](#) are created and then [V2](#) is used as the common ancestor for [16](#) and [17](#).



Now, you have a very good understanding of how Plastic solves even complex merge situations. Admittedly, you won't be facing recursive merges very often, but when they happen it is good to have a proper understanding of what is going on.

A last note: do you remember the entire explanation in "[Merging trees](#)"? In essence, Plastic is merging three trees, and that's why a solution like the one described here is possible. The "tree merge" code takes three trees and then produces a result. All we have to do is to give it a different tree, in this case a virtual one, so it can operate and produce a meaningful result.

Git can do recursive merges too

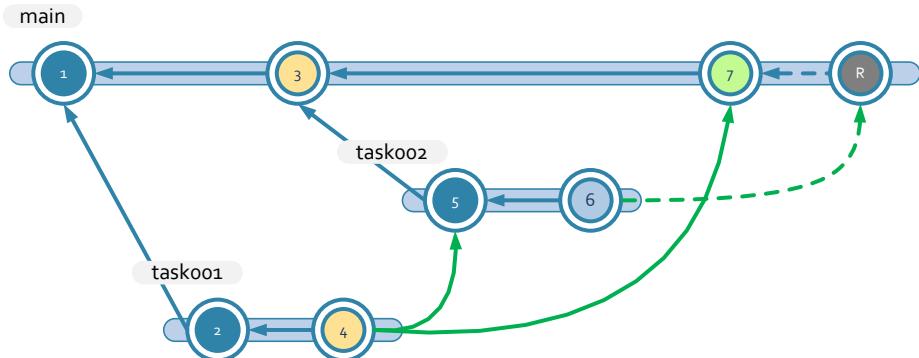
Git implements an algorithm to deal with recursive merges, same as we do. In fact, we tested all our merge cases with Git when we implemented our original solution, to double check we were on the right path.

Recursive merge – why it is better than just choosing one ancestor

You might be wondering: "why all this trouble and complexity instead of just picking one of the possible ancestors to solve the merge"? Ok, let's see why with an example.

In the following scenario, there are two possible common ancestors at the same distance of contributors **6** and **7**. As you can see, this scenario is not a crisscross, it is a more realistic one, but again multiple

ancestors at the same distance are found.



Let's first explore what happens if we randomly choose changeset 4 as common ancestor and perform the merge. By the way, I'm using the following image to introduce the actual contents of the file in conflict in the example.

	6	4	7
10	class Program	class Program	class Program
20	{	{	{
30	static void Main()	static void Main()	static void Main()
40	{	{	{
50	Print("See you");		
60	}		
70	}	}	}

As we learned while delving into 3-way merge, this would be a merge with manual conflict resolution since the line 40 was modified differently by the two contributors. So, choosing 4 as common ancestor will end up in a manual conflict.

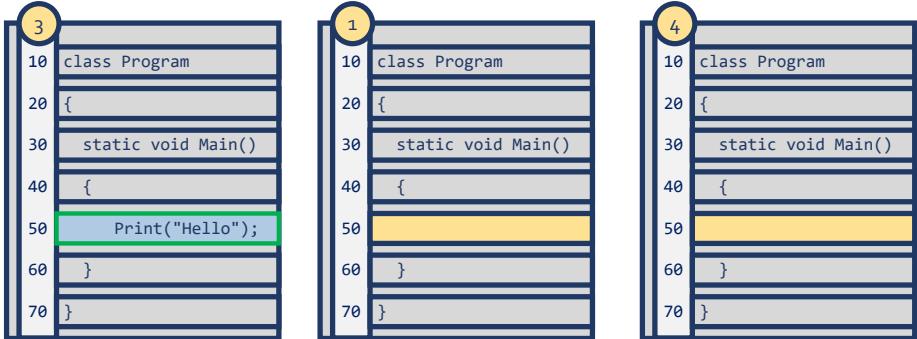
Let's now check what happens if we choose 3:

	6	3	7
10	class Program	class Program	class Program
20	{	{	{
30	static void Main()	static void Main()	static void Main()
40	{	{	{
50	Print("See you");	Print("Hello");	Print("Hello");
60	}	}	}
70	}	}	}

This time, the conflict would be automatic and the result would be keeping the `Print("See you");` sentence.

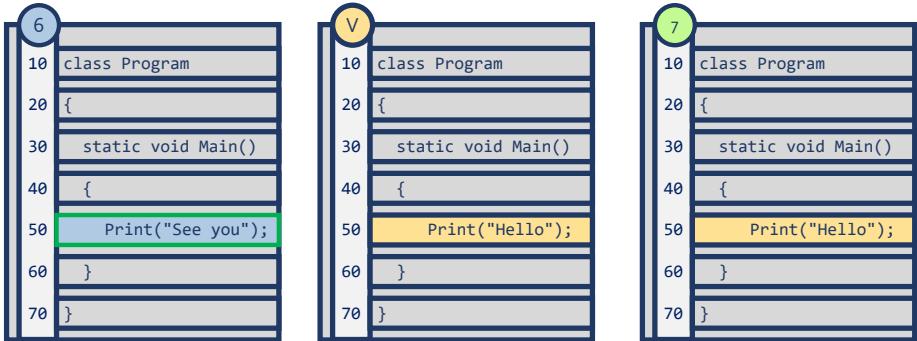
Which one is better? The thing is that without recursive merge you have to rely on good luck. Depending on what the algorithm does, it could pick **3** or **4** and then you can end up with conflicts or even with a wrong automatic resolution. I mean "luck" because there is no heuristic to actually make a better choice.

Now, let's explore what recursive merge would do; it will merge ancestors **3** and **4** to create a new virtual one to be used as the common ancestor for changesets **6** and **7**. Ok, let's go ahead:



As you can see, the virtual merge will be automatic; it will choose `Print("Hello")`; in line 40. No conflicts.

Let's now use this result as the virtual ancestor for **6** and **7**:



Now, it is clear that the merge is automatic again, and the final value for line 40 is `Print("See you")`.

Recursive merge can solve this conflict automatically in a consistent way, and the randomness of choosing one selector over the other is gone.

How important this kind of resolution is? Well, consider a real-world scenario with a few hundred files involved. Suppose your branch history leads to a multiple common ancestor scenario. You could be potentially saving tons of manual conflict resolutions by taking advantage of recursive merge 😊

More about recursive merge

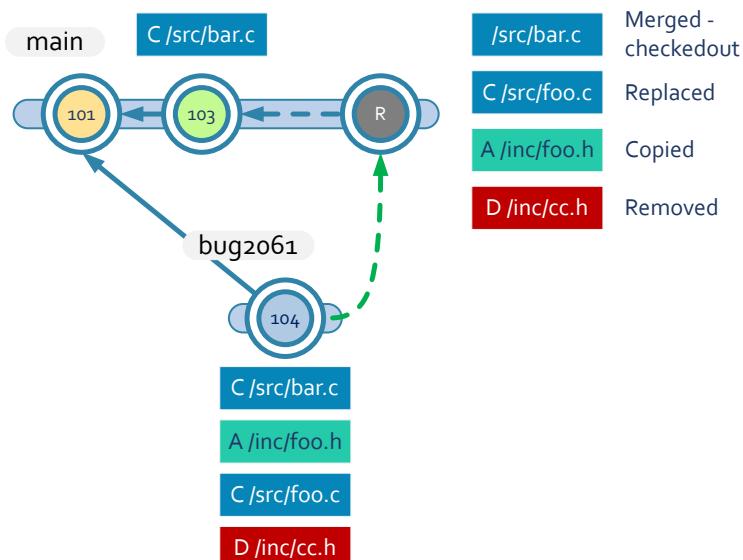
We wrote a couple of blogposts explaining recursive merges a few years ago. I've reused most of the materials for this chapter, with a proper refresh. The original blogpost actually triggered a discussion with the main contributor of Mercurial, were he ranted about Plastic algorithm being broken to finally admitting he was wrong and implementing recursive was "too costly" as the reason why Mercurial didn't have it. You can find the two blogposts here:

- [Merge recursive strategy](https://www.plasticscm.com/download/help/recursivemerge) [<https://www.plasticscm.com/download/help/recursivemerge>]
- [More on recursive merge strategy](https://www.plasticscm.com/download/help/morerecursivemerge) [<https://www.plasticscm.com/download/help/morerecursivemerge>]

Plastic merge terminology

When you merge branches or changesets in Plastic, you'll often find a few names associated with the merged changes before you checkin. Things like "copied", "replaced" and so on. I'm going to explain each of them in this section.

Check the following figure:



- `bar.c` has been modified on both branches, so it will require a merge.
- `foo.h` is added in `bug2061`.
- `foo.c` already existed in `main`, and was only modified on `bug2061` so there won't be any conflicts.
- `cc.h` was deleted in `bug2061`.

Now, once you merge, the files are marked as follows:

- `bar.c` – merged – checkedout. This means the file was modified concurrently because it has changes in source and destination. This one is pretty trivial.
- `foo.c` – replaced. Remember the trees we saw in "Merging trees"? Well, `foo.c` revision in destination

has to be replaced by the one in cset 104 (source) to create the result tree. No merge is needed but there is a replacement of one revid by another one. That's why we call it replaced.

- `foo.h` – copied. Added is surely a better description. The reason why we call it "copy" is because the revid is copied from the tree of cset 104 to the result tree.
- `cc.h` – removed. This one is pretty obvious too. The file is simply deleted from the result tree.

Directory merges

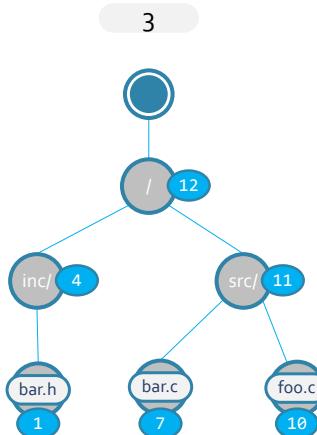
Directories are first class citizens

Directories are first class citizens in Plastic. This means they are versioned in the same way files are. Plastic can have empty directories because they are just not containers of files but entities on their own.



Git, in contrast, can't have empty directories in the tree because directories are not versioned as such, they are just paths needed to host files, but nothing else.

Here is a simple tree of a given changeset:



What does the actual data of revision 10 actually look like? Revision 10 belongs to a file named `foo.c`. Let's see what its associated data looks like:

10	Blob for revid 10
compression: none	
segment: 0	
last: true	
type: data	
	int calc_foo(int x, int y) { return x * y; }

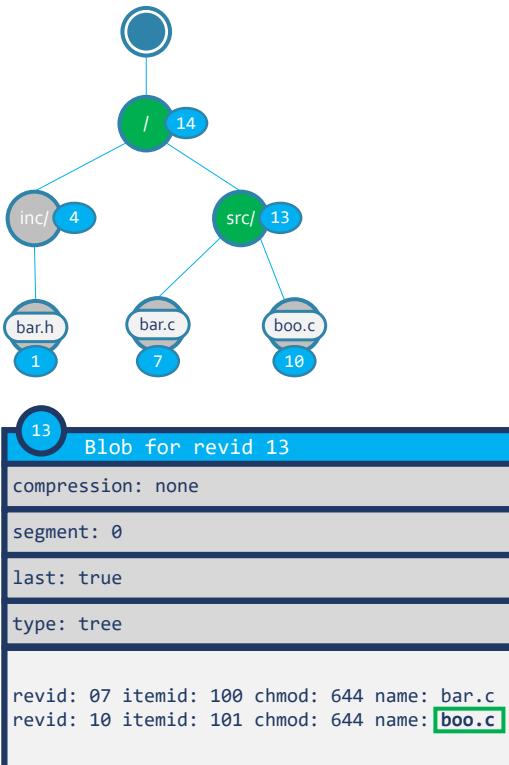
The image shows the contents of the associated *blob* plus its metadata. As you can see, this small fragment of text is not compressed, and since it is very small, it is the segment zero (we split large files in compressed segments of 4MB each).

Now, what is the data of revision 11? This revision is a directory. So, what's the actual data of a directory? You are about to see one of the key data structures that support the Plastic merge power 😊

Blob for revid 11	
compression:	none
segment:	0
last:	true
type:	tree
revid: 07 itemid: 100 chmod: 644 name: bar.c revid: 10 itemid: 101 chmod: 644 name: foo.c	

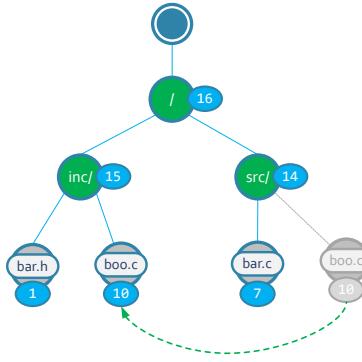
Look at the image carefully. The data of a directory are the names (plus some metadata) of the files it contains. This has a profound implication on how Plastic actually works and merges directories. Files don't have a name, they are not even "aware of" their name, because directories are the ones holding the names. This is going to be key in how directory merges are handled.

Suppose we want to rename `src/foo.c` into `src/boo.c`. What will the resulting tree look like?



Exactly. When you rename `boo.c`, the revision of `boo.c` doesn't change. What changes is its container `src/`. So, a new revision of `src/` is created but not a new one of `boo.c`. In fact, `src/` doesn't know its name either, is its parent (`/` in this case) the one holding its name.

Let's now see what happens if we decide to move `/src/boo.c` to `/inc/boo.c`:



15	Blob for revid 15
compression: none	
segment: 0	
last: true	
type: tree	
revid: 07 itemid: 100 chmod: 644 name: bar.c	
revid: 10 itemid: 101 chmod: 644 name: boo.c	

14	Blob for revid 14
compression: none	
segment: 0	
last: true	
type: tree	
revid: 07 itemid: 100 chmod: 644 name: bar.c	
revid: 10 itemid: 101 chmod: 644 name: boo.c	

Again, `boo.c` won't change, only its source and destination directories will.



In Plastic, when we show the history of a file, we show its moves. This is because we store some extra metadata with info about the moves, just to explain history to users. In reality, the history of a file doesn't really change when you move it to a different location because new revisions aren't created for the file.

Diffing moves

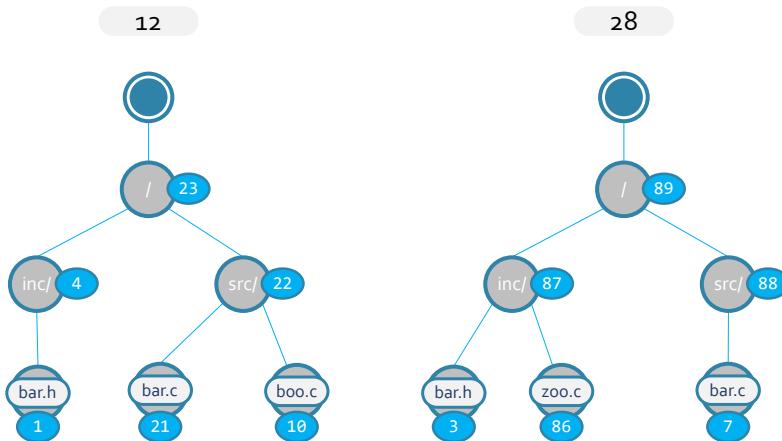
An essential part of merge calculation is diffing trees as we saw in the previous sections. Here, I'd like to explain how moves are calculated after the fact between any two different trees and how the metadata helps find these moves.

Suppose we have a tree similar to the one in the previous section and we want to calculate the diffs between changesets 12 and 28.

The operations that happened in the changes between the two were a move of `/src/boo.c` to `/inc/zoo.c` and then a change in the new `/inc/zoo.c`. I describe the two operations in the figure for the sake of clarity. This doesn't mean both actually happened in changeset 28, they could have happened somewhere in between the two.

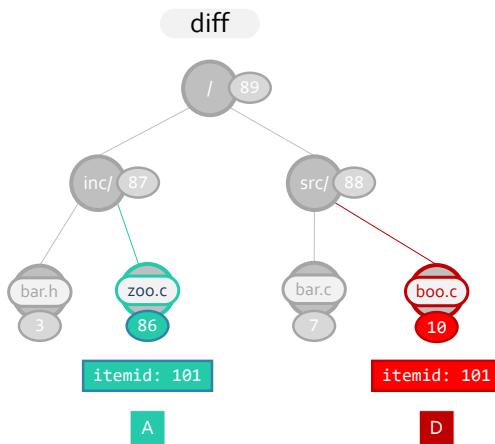
M /src/boo.c /inc/zoo.c

C /inc/zoo.c



When Plastic takes two trees to diff, it starts walking the destination and tries to find matches in the source. Destination is 28 and source is 12.

The actual result is as follows:



Basically, the diff finds that:

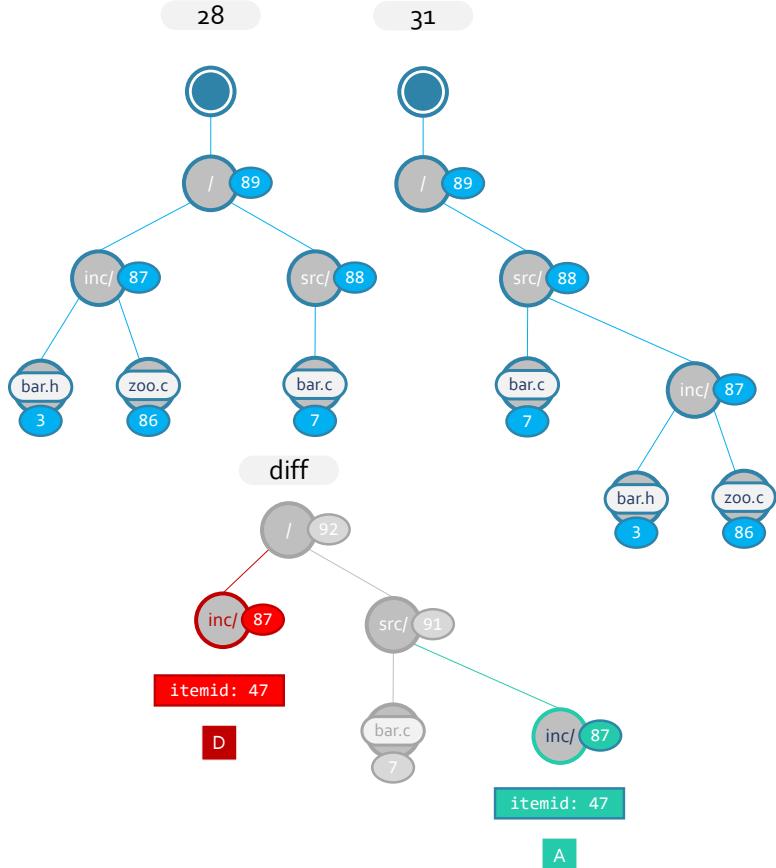
- `/inc/zoo.c` was added.
- `/src/boo.c` was deleted.

The next phase in the diff is trying to pair added and deleted to find moves. In this case, the itemid matches so there is clearly a move between `/src/boo.c` and `/inc/zoo.c`.

Please pay attention to the following: the revision ids don't match. Since `zoo.c` was modified between the changeset 12 and 28, its revid is now 86 while it was revid 10 in the origin `src/boo.c`. So revid alone is not enough to match moves. That's why Plastic uses an itemid to unequivocally identify elements.

The concept of the itemid is crucial in Plastic because it is key for move tracking. It is certainly costly to maintain, but the results pay off because merging moved files and directories becomes straightforward.

For instance, exactly the same logic applies if instead of moving a file we move a directory to a different location:



As you can see in the figure, the diff finds `/inc` as deleted and then `/src/inc` as added but then it can pair them as a move since they have the same itemid. I didn't draw the children of the moved directory in the diff tree because there are no changes in there and the algorithm prunes the walk when there are no more differences. In this case, the diff is even simpler because the revids also match, but of course it would work perfectly too if the revision id of the directory changed.

Plastic vs. Git

The underlying structure of Git is very similar to what I described both here and in "[Directories are first class citizens](#)" except for an important difference: Git doesn't use item ids. It doesn't use revids either, but it uses SHAs (a hash of the content of the file) for the same purpose. What does this mean and how does it impact merge resolution? Consider one of the previous examples where a file was moved, renamed and modified. Git finds the added and the deleted as Plastic does, but then it has to make a guess to match the pairs instead of actually just looking at the itemids. If the file didn't change during the move, then the added/deleted will be easily paired because they have the same SHA. But if the file was modified, then Git has to diff its content to figure out if they still are similar up to a given percentage to conclude they are the same file.

In the case of a directory move, things get more complicated because Git normally has to pair the files inside the directory instead of the directories themselves. In fact, in the example of the directory move, while Plastic shows:

```
mv /inc /src/inc
```

Git will show:

```
mv /inc/bar.h /src/inc/bar.h
mv /inc/zoo.c /src/inc/zoo.c
```

Which can get complicated if instead of two files it was a full tree with hundreds of files in it.

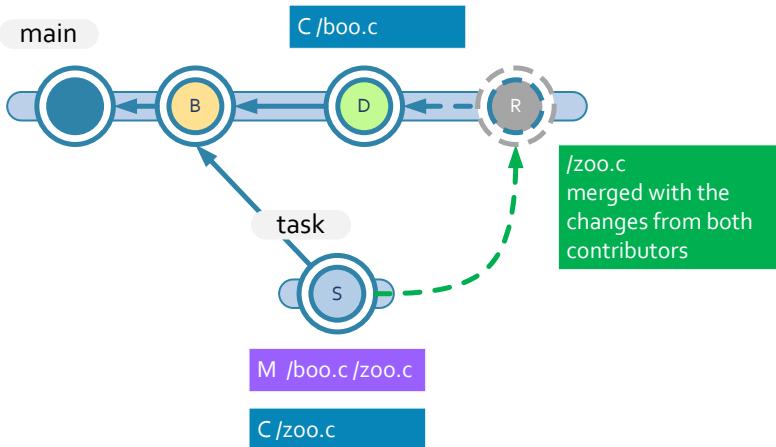
Merging moved files

All the hassle of dealing with itemids pays off to perfect merging moved files.

Moving files and directories is key for refactoring. Not only that, in some programming languages directories are directly associated to namespaces and a change in a namespace can lead to tons of directory renames. It is also very possible that restructuring a project is good to keep the code clean and readable and searchable and ultimately findable.

One approach is to postpone those changes until "we find a moment of calm with fewer concurrent changes". This never happens. The reality is that if you can't refactor your code and directory structure while you continue with normal development, you never will. It is the same as postponing code reviews or any other key practices. You better have the practices and tools to do them on a daily basis or you will postpone them indefinitely.

Well, I hope I explained correctly how important it is to be able to freely and safely move files around. That being said, let's see a couple of scenarios in action.



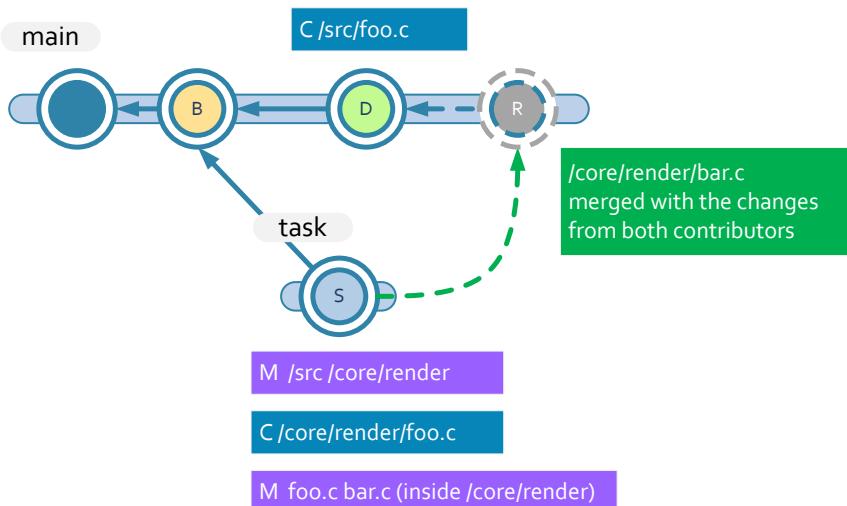
The example in the figure is the canonical "merge a moved file" sample:

- One developer modified `/boo.c`
- While the other renamed it to `/zoo.c` and then modified it

The merge result must keep the rename `/zoo.c` and combine the changes made by the two developers.

In the previous sections, we saw how Plastic diffs the trees to find actual conflicts and a valid common ancestor to merge them. As you know now, the actual names are not relevant for the algorithm since it can identify files and directories by itemid. That's why this case is straightforward in Plastic; it can safely merge `zoo.c` because it knows it is the same item regardless of the names it has on both contributors.

Let's now complicate the scenario a little bit by involving directories.



Let's examine this second scenario:

- The first developer modified `/src/foo.c`
- The second developer first moved and renamed `/src` to `/core/render` (moved inside `/core` and then renamed it from `src` to `render`)

- Then, they modified `/core/render/foo.c`
- Finally, they renamed `foo.c` to `bar.c`.

And again, as convoluted as the scenario might sound, Plastic only finds a conflict in `/core/render/bar.c` because it knows the item was modified by the two contributors. It will run a 3-way merge to resolve possible conflicts while keeping the directory and file moves in the result.

Fear of merge

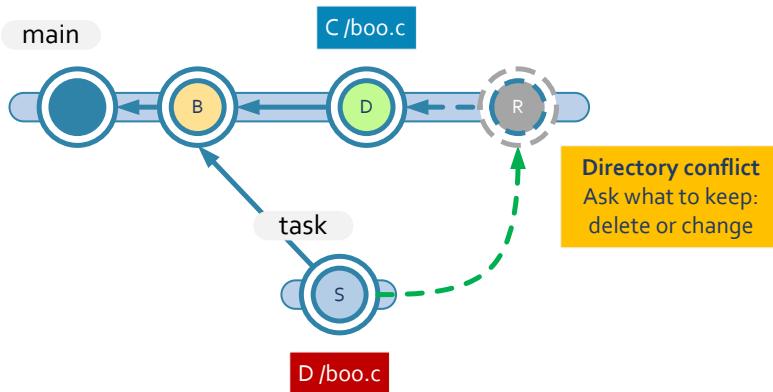
Subversion, CVS, and some other now defunct or very ancient version control systems used to be very bad dealing with moved and modified files. When we made the first demos of Plastic back in 2006 some developers couldn't believe it was safe to rename a file while someone else was modifying it. They had so many bad experiences in the past with SVN that they had already decided to never try anything like this again.

Change/delete conflict

Change/delete conflicts happen when someone modifies a file while someone else deletes it in parallel.

Since the new change can be important, Plastic warns of the situation during the merge so the developer can choose what to do and decide whether the change should be preserved or discarded.

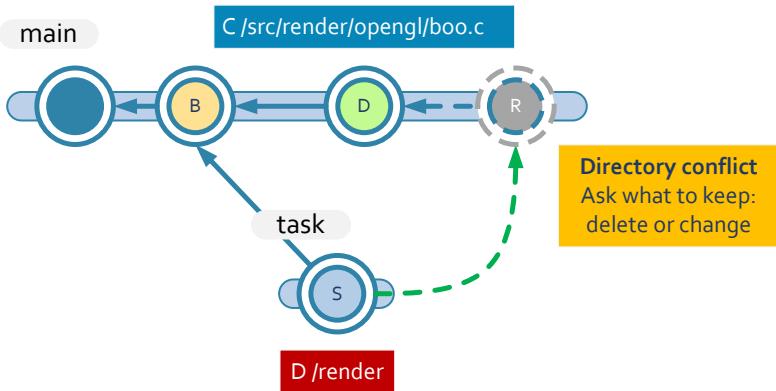
The simplest scenario is as follows:



- Developer 1 modifies `/boo.c`
- Developer 2 deletes `/boo.c`.

Plastic warns of the case during the merge and lets you recover the change discarding the deletion if needed.

A more complex case involves a deeper directory structure and not so direct deletions. Look at the following figure:



- One developer modified `/src/render/opengl/boo.c`
- The other deleted `/render`.

Plastic will detect a conflict and ask you what to do.

Of course, this type of conflict will only be detected if actual elements are in danger of being lost. I mean, consider the following case where we have a directory tree as follows:

```
.
\---src
  \---render
    \---opengl
      |     boo.c
      |     render.c
    \---directx
      render.c
```

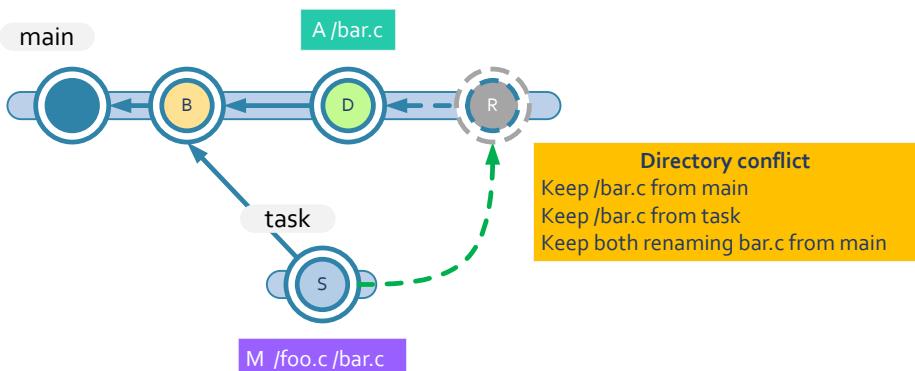
Then, the developer working on the branch named `task` does the following:

```
mv /src/render/opengl /src/opengl +
mv /src/render/directx /src/directx +
rm /src/render
```

In this case, there won't be a merge conflict since `boo.c` was "saved" during the moves done previous to the deletion of `/src/render`, so the changes made in `main` will be applied cleanly.

Add/move conflict

Add/move conflicts happen when you add a file or directory, and concurrently someone moves an existing file or directory to the same location where you added the file. Plastic will help you solve the conflict.



The example in the figure matches what I described before:

- Developer on main adds `bar.c`
- Meanwhile, the developer on task decides to move `foo.c` to `bar.c`.

Here itemids come to the rescue. Plastic is not fooled by the fact the files have the same name. It knows they are different files because they have different itemids. So, Plastic will ask you to keep one or the other or to keep both by renaming one of them.

I won't describe a more complex case with directories because it will be actually the same as the one we just saw. Finally, a combination of files and directories where names collide is also possible (suppose the added `bar.c` was a directory) but the resolution will be exactly the same as described above.

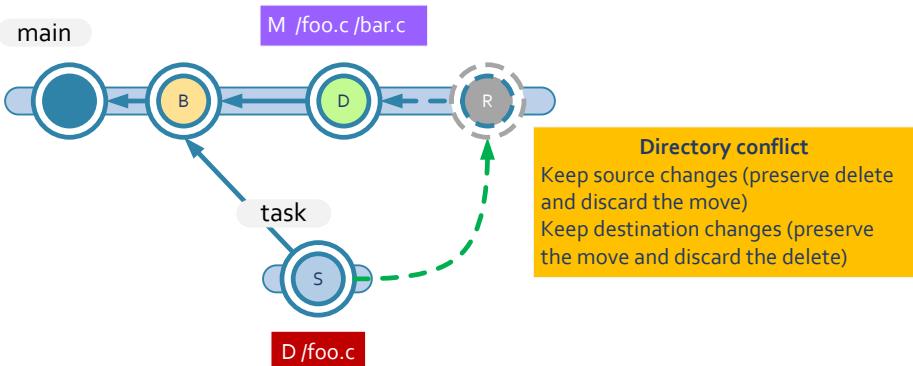
Alternative resolution

There is a possible alternative solution for the first scenario involving the added `bar.c` and the renamed `foo.c` to `bar.c` which is merging the changes to keep a single `bar.c`. It would be a 2-way merge since there is no possible common ancestor. We have never implemented this alternative so far because we consider that the user typically wants to discard one of the files or rename one of them and keep both. But, this is something we considered several times in the past, so if you think you really need this option, please let us know.

Move/delete conflict

Move/delete conflicts happen when you move a file or directory to a different location and concurrently someone else deletes the destination directory. If not handled correctly, a merge of the concurrent changes could lead to valuable data loss.

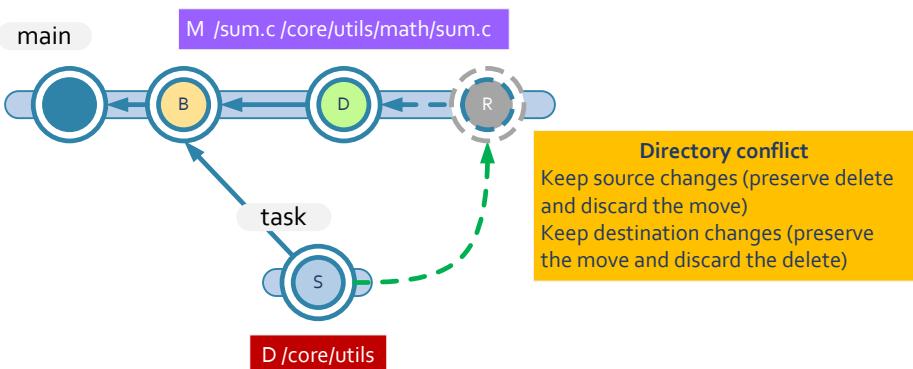
Let's see an example:



- You move `/foo.c` to `/bar.c`.
- In parallel, your colleague deletes `/foo.c` in a different branch.

When you merge his branch into yours, Plastic will warn you about the move/delete conflict and will let you go ahead and remove the file or keep the delete.

This type of conflict makes more sense when the scenario involves directories. Look at the following figure:



- One developer decides to move `/sum.c` into the directory `/core/utils/math` because he thinks the code makes more sense there.
- In parallel, someone did a cleanup and deleted the `utils` directory (possibly moving files outside to refactor the code structure, but of course not "saving" `sum.c` because it wasn't inside `/core/utils` when they performed the cleanup).

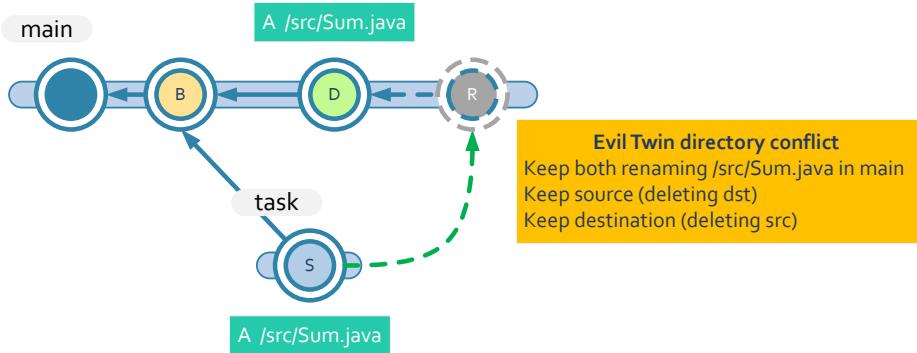
When the branches get merged, Plastic will detect the move/delete conflict and warn you about the situation and let you decide whether you really want to keep the delete and lose the moved file, or undo the deletion.

If you undo the delete of `/core/utils` during merge, chances are you will "restore" many other files and directories inside `utils`. But, they will stay in your workspace after the merge and you will have the chance to delete again what you don't need, but also to preserve `sum.c` which is what you really wanted.

Evil twin conflict

Evil twin conflicts happen when two different items collide in the same location. To be honest, I don't remember anymore where the dramatic name of "evil" came from. Not sure if we took it from Clearcase, if we found it somewhere else or if we simply came up with it.

Anyway, the conflict is very simple to understand with an example:



- Developer 1 adds `/src/Sum.java` in branch `main`.
- Meanwhile, developer 2 also adds `/src/Sum.java` in branch `task`.

The merge will detect an evil twin conflict because the directory can't have two items with the same name in it.

Plastic proposes these resolution options:

- Rename `/src/Sum.java` in destination (`main` in this case) to a different name and preserve the two files.
- Keep the file coming from source, deleting the one that was already in destination.
- Keep the file coming from destination ignoring the one from source.

Do you miss a 2-way merge and unify history?

There could be a different possibility to solve this conflict which is to unify somehow the contents of `/src/Sum.java`. We could keep the itemid from the file created on `main`, but merge the contents with the file coming from `task`. It would be a 2-way merge since there is no possible common ancestor.



We considered this option several times over the years and never end up implementing it since the default action users normally take is to keep both because they realize they really wanted to have two different files and the name collision was not on purpose.

We will keep an eye on this conflict resolution and eventually add 2-way merge as a solution if users find it useful.

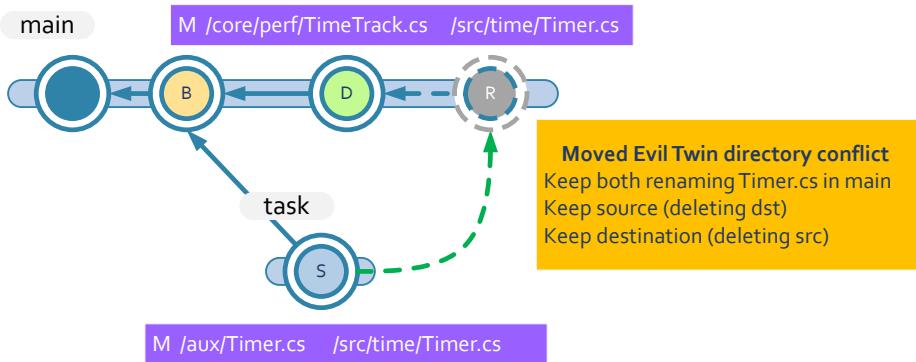
As a temporary workaround, it is always possible to keep the two files, then use a merge tool capable of doing 2-way merge (like Kdiff3), merge them, then delete the file you don't want before checkin.

Moved evil twin conflict

A moved evil twin conflict happens when you move two files or directories to the same location in different branches and then you try to merge. The directory can't contain two items with the same name

and a conflict is raised.

Check the following example:



- One developer decided to move `/core/perf/TimeTrack.cs` to `/src/time/Timer.cs`.
- Meanwhile, on a different branch, a different developer moved `/aux/Timer.cs` to `/src/time/Timer.cs`.

This scenario usually happens when some similar functionality exists in two different places (like the case here).

In this type of conflict, Plastic provides three options:

- Keep both files by renaming the one on destination, `main` in this case, to a different name.
- Keep only the one coming from source deleting the one in the destination.
- Keep the one in destination ignoring the one on source.

As with regular evil twins, it could be interesting to have the choice to do a 2-way merge if we know the files are really the same although created separately. If you think this is something interesting, please let us know.

Itemids and Evil Twins

As you know already, Plastic identifies each file and directory by an itemid. The itemid is key for calculating diffs and correctly tracking moves. Most of what makes Plastic merge more powerful than Git and other version control systems lies in the precise tracking allowed by itemids.

There is a downside, though. Every single directory/file you add to version control has its own itemid even if you mistakenly added the same file twice and you really would like it to be just a single file. Consider the typical evil twin case where you do as follow:

1. Add `/AssemblyInfo.cs` in `main/task001`
2. Add the same `/AssemblyInfo.cs` in `main/task002`

When you merge the two branches, you probably expect Plastic to be clever enough to know they are both the same file and that it doesn't make any sense to consider them as different items. But Plastic throws an evil twin conflict.

There will be cases where this is exactly the behavior that you expect and appreciate, because the team mistakenly added two variations of the same file in the same location.

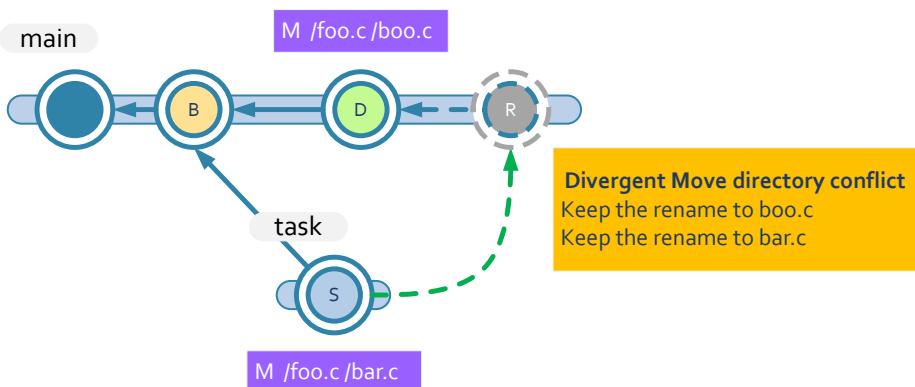
But there will be others where it will simply hit you.

We are considering adding both a 2-way merge solution for this case, and also a "unify item" where after the merge the histories are somehow unified (which is not that straightforward because it would mean rewriting previous history).

Divergent move conflict

This is my favorite merge conflict and the one I often use in demos because I think it really demonstrates the strength of the Plastic merge engine.

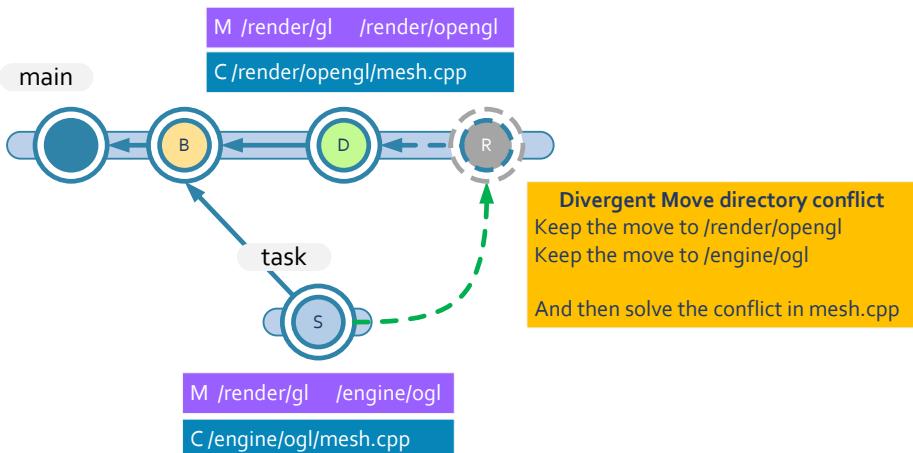
As usual, let me explain it with a graphic:



- One developer decides to rename `foo.c` to `boo.c`
- In a different branch `foo.c` is renamed to `bar.c`

Plastic detects the conflict on merge and asks you what to do, which name do you actually want to keep.

Things can get more complex if we involve directories together with concurrent changes on a file:



- Here first `/render/gl` was moved to `/render/opengl`, and then `mesh.cpp` modified
- In parallel `/render/gl` was moved to `/engine/ogl` and again `mesh.cpp` modified

Plastic will first ask you to decide where to put the `gl` directory. Do you want to rename it and keep it under engine? Or move it to /engine and rename it to `ogl`?

And, once the directory conflict is solved, Plastic will ask you to merge the `mesh.cpp` file.

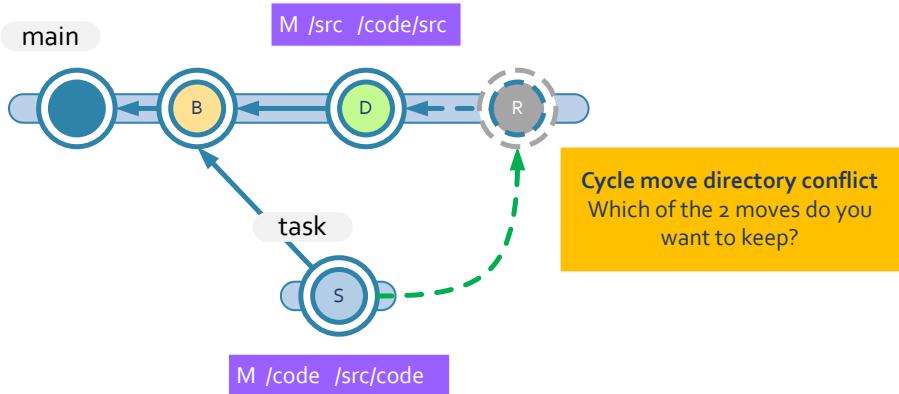
Why do I like this type of conflict so much? Well, because it really turns a complex (although obvious) scenario into something fully manageable while other version control systems (Git among them) will turn this into a complete nightmare. What Git does is duplicate the directory and all its contents, which makes unifying `mesh.cpp` extremely complex and unlikely to happen, which normally leads to losing changes.

Cycle move conflict

Cycle moves are definitely not common, but they can happen if you move one directory inside a different one (`src` into `code/src`) and in the meantime somebody else performs the opposite operation (`code` into `src/code`).

Imagine we have this tree

```
/  
/src  
/code
```



What Plastic does when it detects this case is to ask the user which move to keep, because keeping both is not possible.

Internally, it could create some sort of infinite loop where one is child of the other recursively (not good).

What some version controls do is to duplicate the trees, so you end up with `/src/code` and `/code/src` and all the files duplicated. It doesn't look like a good conflict resolution strategy 😊

Conflict resolution order – directories before files

When you solve a complex merge in Plastic, you'll always have to solve directory conflicts (structure conflicts if you prefer) prior to file conflicts.

This is because you need to have a valid structure prior to "plugging" the files into it. So, if the directory structure needs you to solve whether you want to keep the rename from `bar.c` to `foo.c` or `boo.c`, you first have to solve that prior to merging the actual content of the file.

Both the command line and the GUIs follow this order to solve conflicts.

The following screenshot taken on Windows (macOS and Linux follow a quite similar pattern) shows a divergent move directory conflict followed by a file conflict on the same file; you first have to solve the directory conflict and then merge the file.

1 directory conflict Conflicts that might change the directory structure (added, moved, deleted). You must resolve these conflicts before processing merges.

1 conflict pending to resolve.

Conflict	Item name	Resolution method	Status
Divergent move conflict	/kernel/FileSystem.cs	Choose...	Unsolved conflict

1 items to merge For 1 items, both source and destination contributors have changes. User interaction might be required (manual merge)

Item name	Remarks	Contributor
C Changed on both source and destination, might require a manual merge (1 items) /kernel/FileSystem.cs	Changes in both source and destination contributors	Merge changes from both contributors

[Merge options...](#) [Recalculate merge](#)

Automatic resolution of directory conflicts

For each type of directory conflict choose whether the source or the destination contributor should be automatically selected to resolve the conflict. See the following example:

```
cm merge cs:2634 --merge --automaticresolution=eviltwin-src;changedelete-src
```

The above merge operation from changeset 2634 resolves the `eviltwin` and `changedelete` conflicts by keeping the source (`-src`) contributor in both cases.

- A `-src` suffix after a conflict type tells the merge command to keep the source contributor changes.
- A `-dst` suffix will keep the destination contributor changes.

Find below the complete list of conflict types the merge command supports:

- `movedeviltwin`
- `eviltwin`
- `changedelete`
- `deletechange`
- `movedelete`
- `deletemove`
- `loadedtwice`
- `addmove`
- `moveadd`
- `divergentmove`
- `cyclemove`
- `all`

The `all` value overrides the other options. In the following example, `eviltwin-dst` will be ignored:

```
cm merge br:/main/task062 --merge --automaticresolution=all-src;eviltwin-dst
```

Merge from and merge to

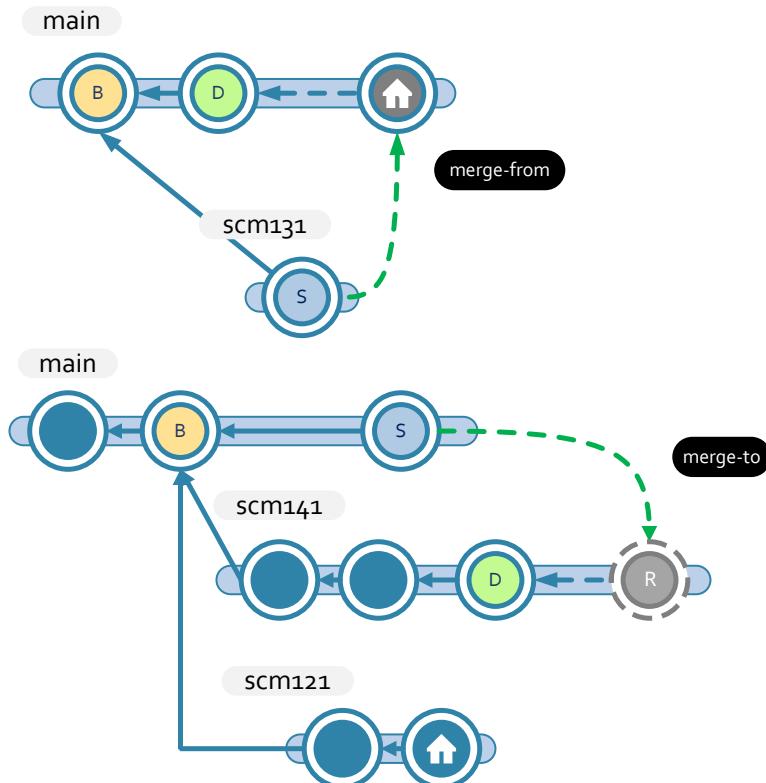
The typical way to do a merge in version control is as follows: you put your workspace pointing to the result, and then you **merge from** source. That's why we call it "merge from".

It is the most common type of merge because it involves you, as the developer, to get the merged changes into your workspace and review them and test them before you checkin.

But, there is a different type of doing merges in Plastic that doesn't require a workspace at all; it can be entirely done on the "repository side". It is useful under many different circumstances but especially with CI systems. I mean, years ago it was very common for the developer to test his own changes after doing a merge and before doing the checkin. Today, thanks to the ubiquity of CI and DevOps, it is more common to find a bot of some kind performing the merges. In those cases, having a workspace can be even a burden because it means having extra working copies. By the way, the workspaces will be needed to build the code, but the merge step can be triggered by a different machine where having a working copy would be just an extra burden.

"Merge to" is also very useful as developers when you want to solve a merge conflict without having to switch to the destination branch, which can save quite a bit of time.

The figure shows the key differences between the "merge to" and "merge from" operations. We also call "merge to" a "workspace less" or "server side" merge.

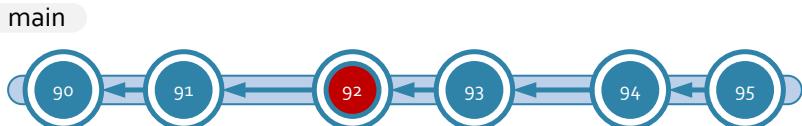


Removing changes – subtractive merge

What is a subtractive merge

Subtractive merge is a very powerful merge operation that allows you to remove a given change while preserving the rest of the changes made later.

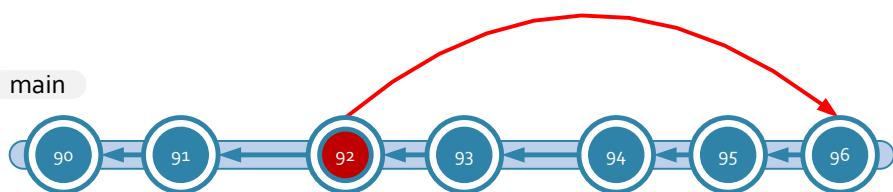
Consider the following scenario:



You need to get rid of the changes made in changeset 92, but the ones done later are perfectly fine.

Deleting 92 is not an option since, as we saw in "[Delete empty branches only](#)" it is only possible to delete 92 if you delete 93, 94 and 95 too.

Subtractive merge comes to the rescue here as follows:



$$96 = 91 - 92 + 93 + 94 + 95$$

By "subtracting" 92 you create a new changeset 96 which only removes the changes done in 92, but keeps the ones done later.

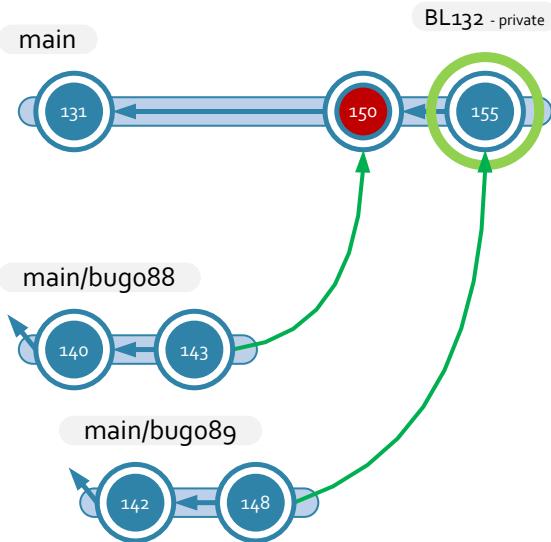
When should you use subtractive merge?

Simple rule of thumb: don't abuse subtractive merge. If you find lots of red lines in your repo (that's how we render subtractive), be very careful. Subtractive is a tool to handle exceptions, not something to use on a daily basis.

The reason for this is because subtractive merge links are informative: they help you understand what happened, but they are not actively considered for merge tracking. And, you already know [why merge tracking is so important](#) to avoid issues. So, use it wisely.

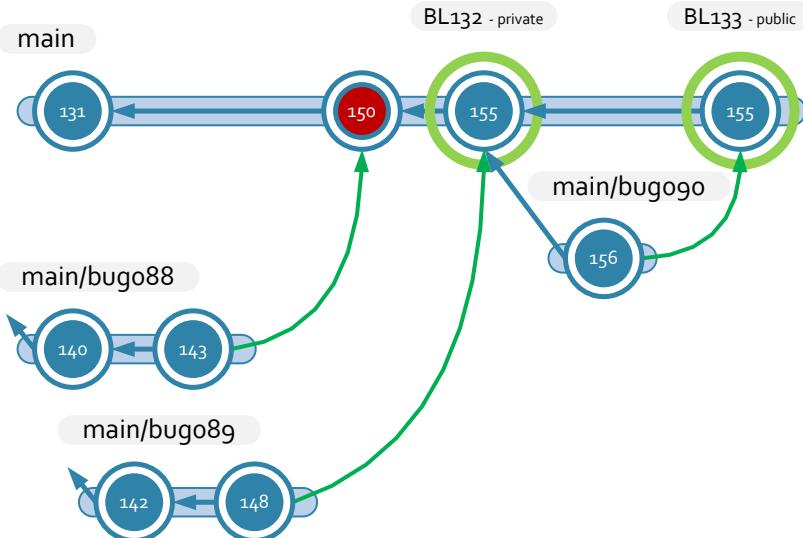
How to undo a merge

Consider the following scenario: a couple of new branches were merged to create a new version, but then after some more intensive testing or during internal deployment, you detect the release is broken and figure out that branch `bug088` is the one that introduced the bug:



How can you proceed here?

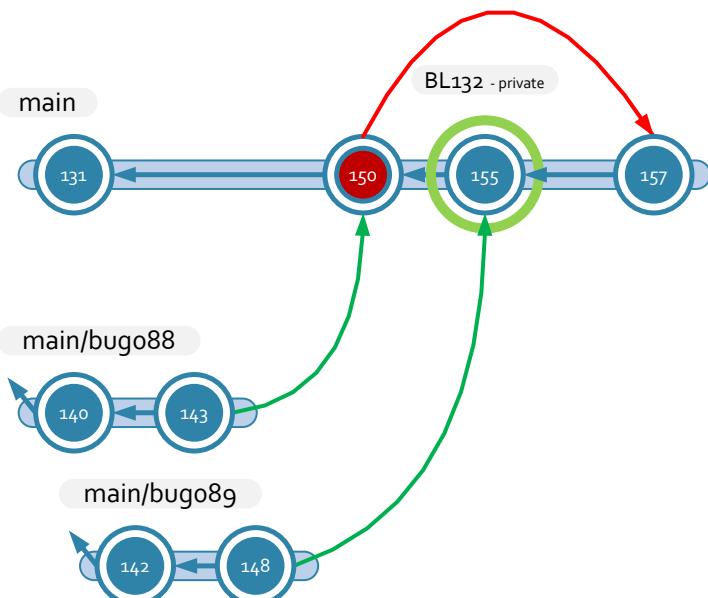
My recommendation, under normal circumstances, would be to implement a new task to fix the issue and solve the broken build:



I like this solution because it doesn't introduce an exception case, it simply sticks to the same "task branch" and regular DevOps cycle. There is no manual intervention needed, just a new task branch that will be tested and merged by the CI system or a Plastic mergebot and everything will proceed as usual even if we are fixing an unusual emergency provoked by a broken build.

Well, but I'm in the "subtractive merge section" so you most likely expect me to use a subtractive merge to solve the scenario, right?

Here we go:



As you can see, the other alternative is to simply take advantage of subtractive merge to remove the changes made on [bug088](#) (we are subtracting the changeset result of its merge, so all its changes will be gone) while preserving the later [bug089](#) which was perfectly fine.

Merging again a branch that was subtracted

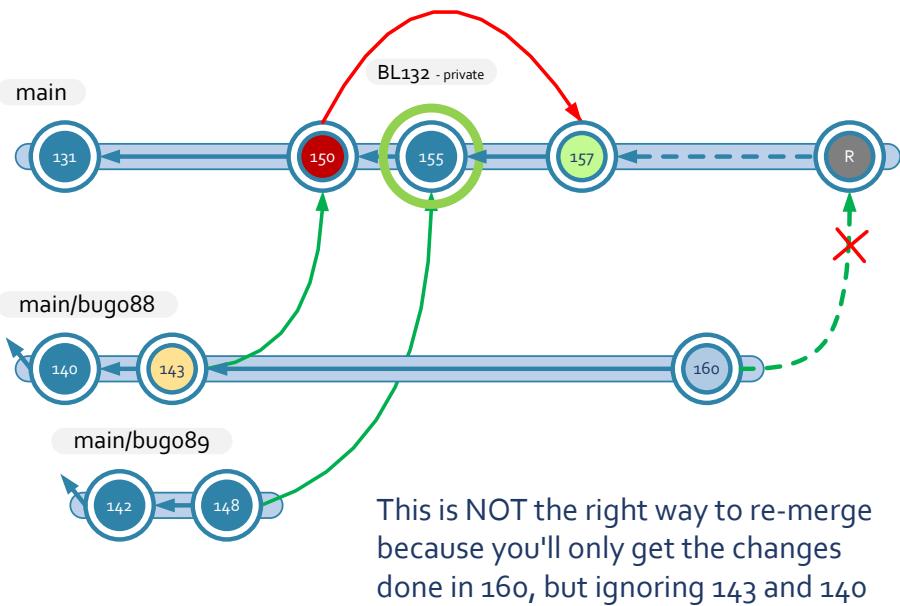
Subtractive is powerful, but it is definitely not a beginner's tool.

In the purest "task branches" philosophy, you would abandon [bug088](#) and create a new task to fix the new issue. But, sometimes that's not doable because it is better to simply fix whatever happened to [bug088](#) and merge again.

And it is the "merge again" that is worth a section, since it is far from straightforward.

Let's continue with [bug088](#) and fix the issue we introduced there just by doing a new checkin on that branch. Then, it would be possible to simply merge it back, right?

Wrong.



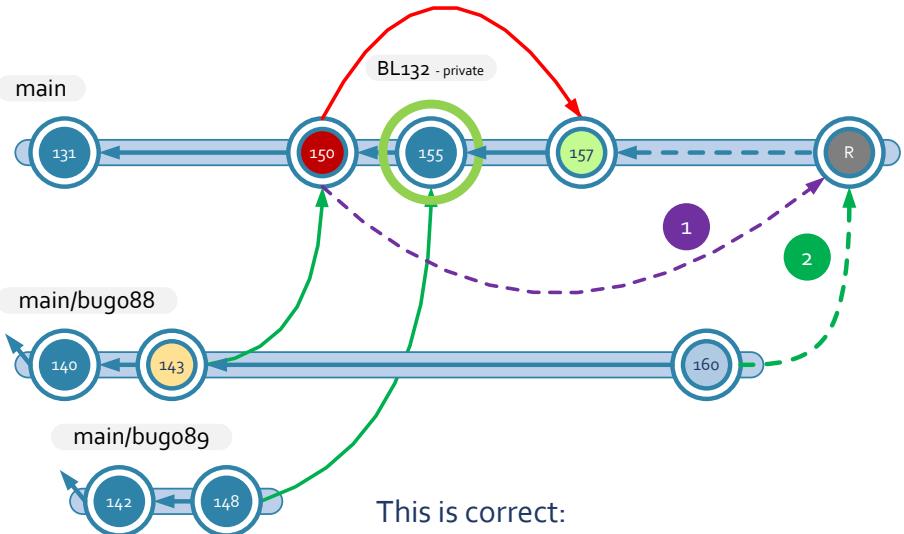
Now, you have a good opportunity to put in practice all the newly gained expertise in merge tracking. What happens when you merge 160 into 157?

Well, first 143 will be the common ancestor, and as such, only the changes made in 160 will be merged. It means you are not really introducing all the changes done in `bug088`, but only the new changes in 160.

This is because the subtractive merge link is informative and not part of the merge tracking calculation. So it is completely ignored.

What is the right way to do it then?

Pay attention to the following figure:



This is correct:

- 1) Cherry-pick 150 to reintroduce the changes => **ignoring traceability!**
- 2) Now merge from bugo88 again

To "reintegrate" a previously subtracted branch with newer changes you have to:

1. Cherry pick the subtracted changeset to reintroduce the changes. Important: ignore traceability in the cherry pick or the merge won't happen. This is because you are cherry picking from 150 to a child changeset of 150... so normally the merge wouldn't have to happen.
2. Then merge the branch again.

The procedure is not hard, but you must know what is going on to properly use this feature. I hope the explanation helped you master subtractive.

Cherry picking

Cherry picking is a way to merge just the changes made on a changeset or set of changesets instead of using the regular merge tracking to decide what needs to be merged.

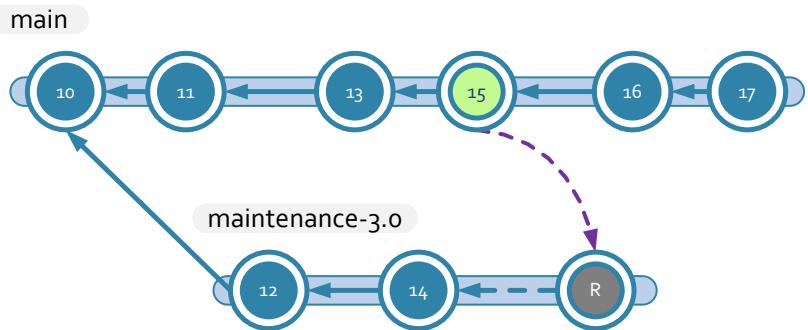
There are several variations of cherry picking that we'll be seeing in a minute.

Cherry picking doesn't alter merge tracking, so be very careful and don't overuse it because it is again a tool to handle exceptions, not to be used on a daily basis.

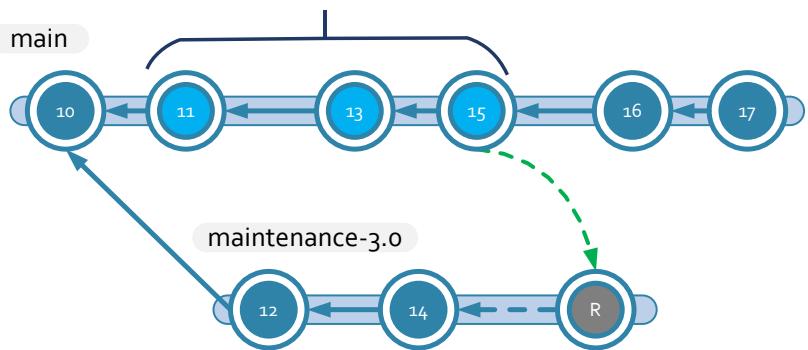
Cherry pick a changeset

When you cherry pick a changeset, it means you just want to apply the changes made in this changeset to a different branch. It is like applying a patch just taking the changes made in the changeset.

The following figure explains the case: by cherry picking changeset 15 to branch `maintenance-3.0` you are just applying the changes made in 15 to the branch.



What would happen if you merge from changeset 15 instead? This is very important to differentiate because very often users get confused about the difference between cherry pick and merge. Look at the following figure:

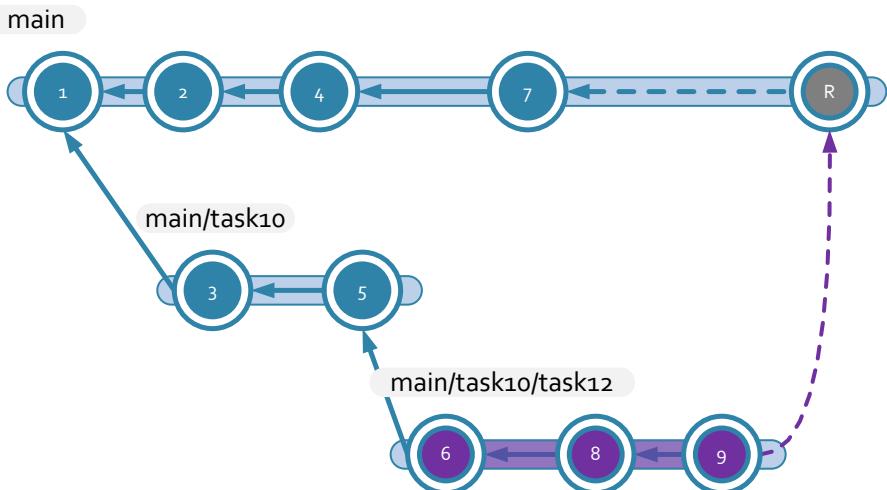


A merge from changeset 15 would apply the changes made in 11, 13 and 15, and not just the ones in 15 as cherry pick does. That's the key difference between cherry pick and merge.

Branch cherry pick

We can also apply a cherry pick of a whole branch instead of just a changeset.

Consider a scenario like the following: we want to cherry pick `main/task12` to `main` instead of merging it.



The cherry pick will take the changes made on changesets 6, 8 and 9 into `main`, while the merge would also bring the changes made in 3 and 5 in the parent branch `main/task10`.

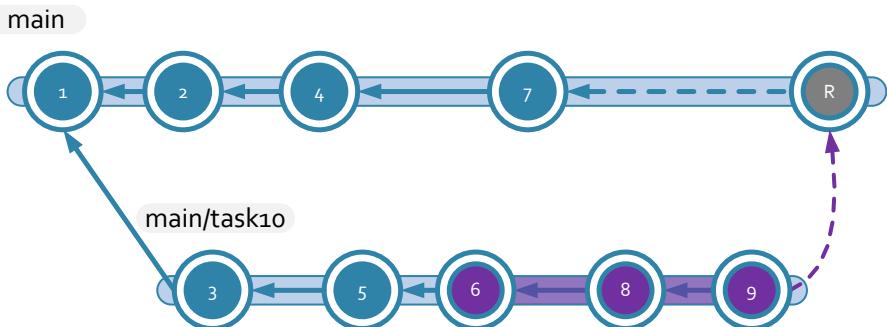
To put it in more formal terms (which I bet some of you will appreciate), the merge in the figure considers the `(5, 9]` interval; changesets 6, 8 and 9 will be "cherry picked".

Branch cherry picking can be very useful under certain circumstances when you just want to pick a few changes instead of the whole history.

Interval merge

Branch cherry pick is indeed a subset of the interval merge, where the interval is set by the topology of the branch.

The figure shows an interval merge of `(5, 9]` which means the changes made in 6, 8 and 9 will be cherry picked to `main`.

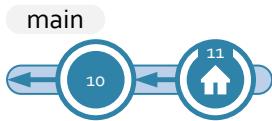


Conflicts during checkin – working on a single branch

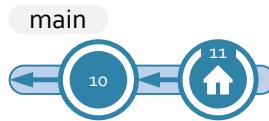
There are cases where teams choose to work on a single branch because of different reasons. It can be a pair of developers collaborating on a single branch at the early stages of a new project or feature, or maybe the entire team working together on `main` because they have unmergeable files that they need to lock.

Let me show you how a possible scenario might evolve:

Ruben working on `main@fly@skull:9095`



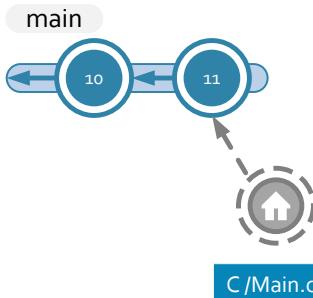
Dani working on `main@fly@skull:9095`



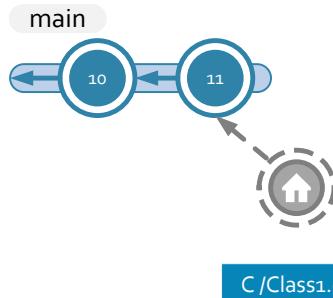
It all starts with Ruben and Dani working on the `main` branch. Each of them works on their own workspace, so they have independent local working copies. And they are both synced to changeset `11`. The houses represent where their workspaces are.

Then both Ruben and Dani start making changes. As you can see, they modified two different files.

Ruben working on `main@fly@skull:9095`



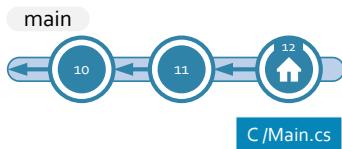
Dani working on `main@fly@skull:9095`



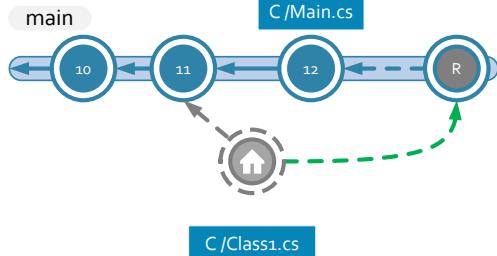
Ruben is the first to checkin. He created changeset `12` when he checked in the change to `Main.cs`.

Now, Dani is ready to checkin `Class1.cs` too, but the repo is different to what Ruben faced: there is a new changeset `12` that is in potential conflict.

Ruben working on main@fly@skull:9095



Dani working on main@fly@skull:9095

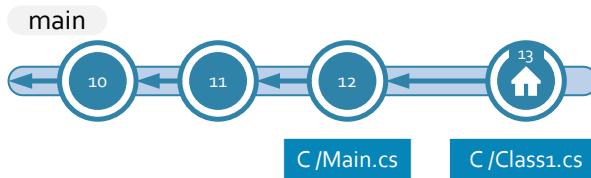


The Plastic merge algorithm finds a potential conflict from the "topological" point of view since the changeset 12 is now in `main` and can collide with the local changes made by Dani.

In this case, the merge algorithm is smart enough to find out that the changes don't collide; they changed different files.

Then, the checkin performs what we call an "**update-merge**"; the workspace of Dani is updated to latest during checkin, so his home would virtually go to 12, but then his changes (that do not collide at all with the ones coming from `main` at 12) are kept as if they were done on 12 (it is not a Git rebase at all, but some of the ideas behind are similar as you can see in "[Plastic rebase vs. Git rebase](#)").

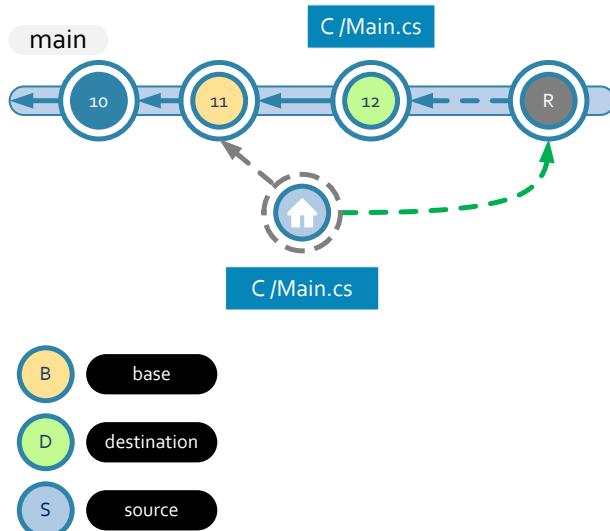
Dani working on main@fly@skull:9095



The "update-merge" leaves a linear history without merge links, something we found is best in this scenario since users working on a single branch normally get confused by merge lines.

Now, what would happen if they had both modified `Main.cs` instead? The following figure illustrates the case:

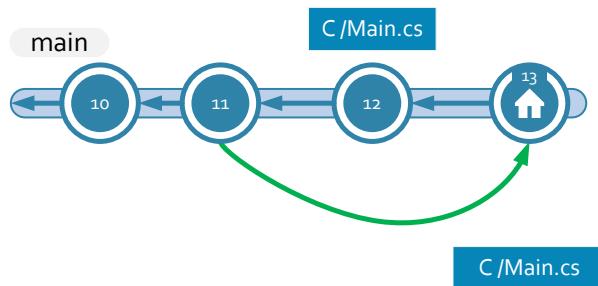
Dani working on main@fly@skull:9095



Now, `Main.cs` requires a real merge, so the regular merge algorithm takes control. As you can see, we have a base, and then destination (12) and source (our local change) contributors, like in any regular merge.

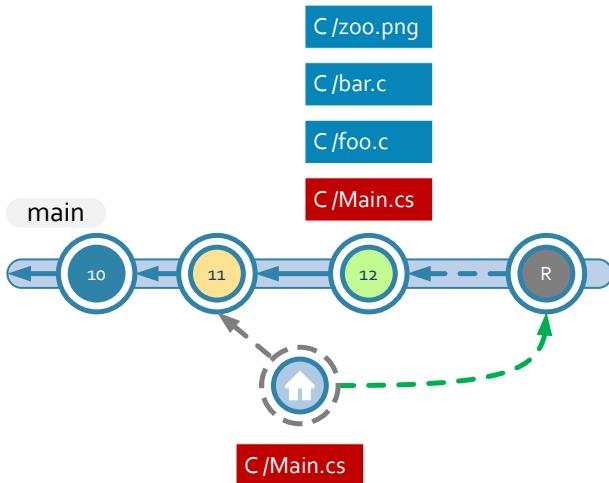
Once Dani checkins the result of the merge, the Branch Explorer will be as follows:

Dani working on main@fly@skull:9095



And this can be confusing to some users who don't really understand why a merge link happens inside a single branch. But, the reason is clear. A merge from 11 + some changes happened to create 13.

Finally, I'd like to explain what happens when the merge during checkin involves more files than just the ones in conflict, because this is a subject that causes confusion quite often. Check out the following scenario: my local change is just `Main.cs`, which was also changed by someone else in changeset 12. But, in changeset 12 other three files were changed. They are not in conflict at all, but they need to be downloaded to my workspace as part of the merge.



The sequence of steps is as follows:

- You want to checkin and then Plastic warns you there are colliding changes.
- Then you go and merge `Main.cs`.
- You finish the merge and then check the status of your workspace.
- And then you see `zoo.svg`, `bar.c` and `foo.c` as "checkouts" in your workspace.

"Why are they marked as "checkedout" if I never touched these files" – some users ask.

Well, you didn't change them, but they were involved in the merge, that's why they are marked as checkouts. For more info about the status of the files during a merge, see "[Plastic merge terminology](#)".

Simplifying merges on a single branch

We have plans to greatly simplify merges on a single branch, so expect changes in the way they are currently handled.

Many users reported that seeing merge links in merges that happen as a result of a checkin on a single branch are confusing, and they would prefer not to see them.

Our goal is to do the following:

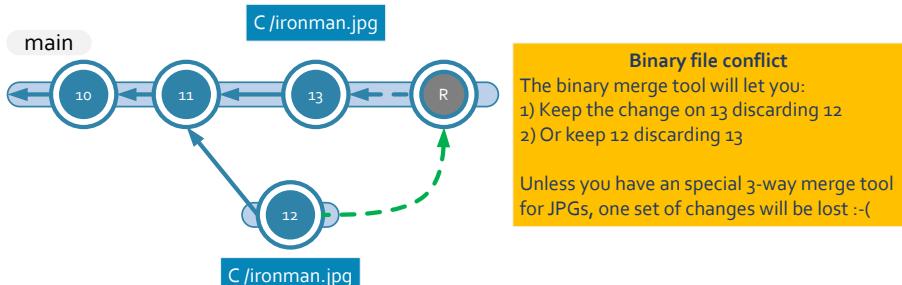
- If a merge is needed during checkin, merge the involved files but alter merge tracking. The destination will be always the head of the branch at the moment of the merge and the base will be the parent of the local changes.
- Do not show the newly downloaded elements as "part of the merge" (the files coming from the head of the branch) since it is often very confusing for users.

We already started part of this effort with Gluon, the GUI and CLI to deal with partial workspaces. Check [this blogpost](#) [<https://www.plasticscm.com/download/help/gluoncanmerge>] for more info.

Locking: avoiding merges for unmergeable files

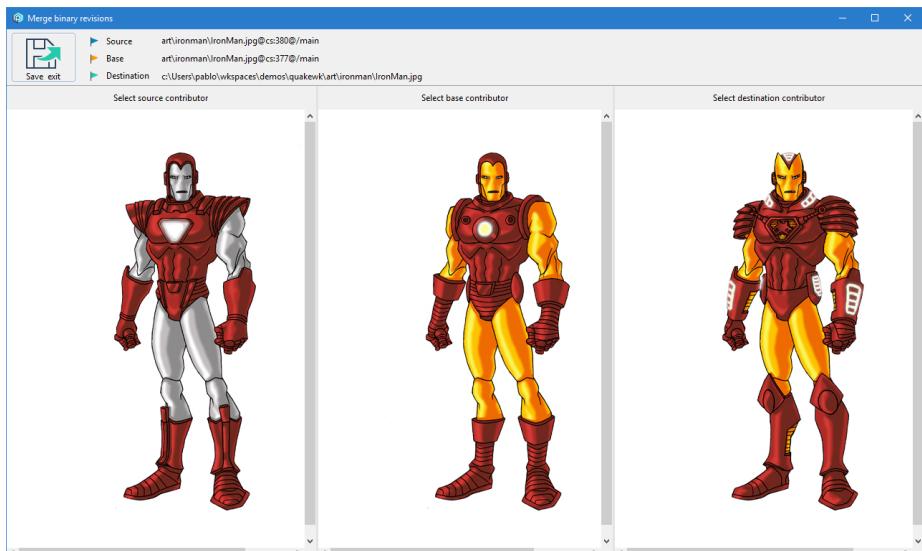
Merging binary files

What happens if two team members modify an unmergeable file concurrently? Suppose you have a `.jpg` file and two artists made a change to it concurrently.



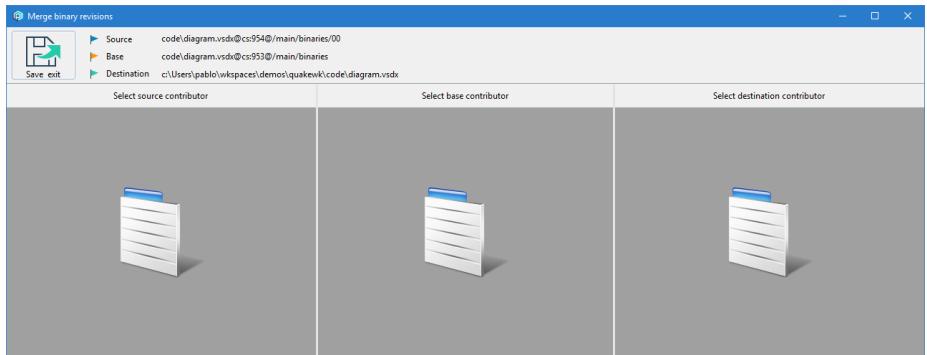
Plastic comes with a binary merge tool that is invoked only if a file is detected as binary (Plastic bases its decision on the file extension for known types, the `filetypes.conf` config file, and an algorithm that reads the first bytes of the file and tries to figure out if it is binary or text. There is also a way to configure a mergetool based on the file extension). So, in this case, since `ironman.jpg` is detected as a binary, it will invoke the binary merge tool which will only let the user choose which contributor to keep and which one to discard.

Since it is a jpg file, the binary merge tool can show a preview to help you with the decision as follows:



Anyway, while it shows you the common ancestor and the two contributors, it is unable to actually merge the pictures to produce a new one, and one of the changes will be discarded.

In case the file in conflict was not an image, binary merge tool won't be able to create a preview (unless you plug a preview generator, but that's a [different story](#) [<https://www.plasticscm.com/download/help/custombinarypreview>]). What you would see in this case would be something as follows:



Anyway, what is clear from this example is that while there is a way-out during merge, it is not a good idea to discard work trying to work in parallel in files that can't be merged.

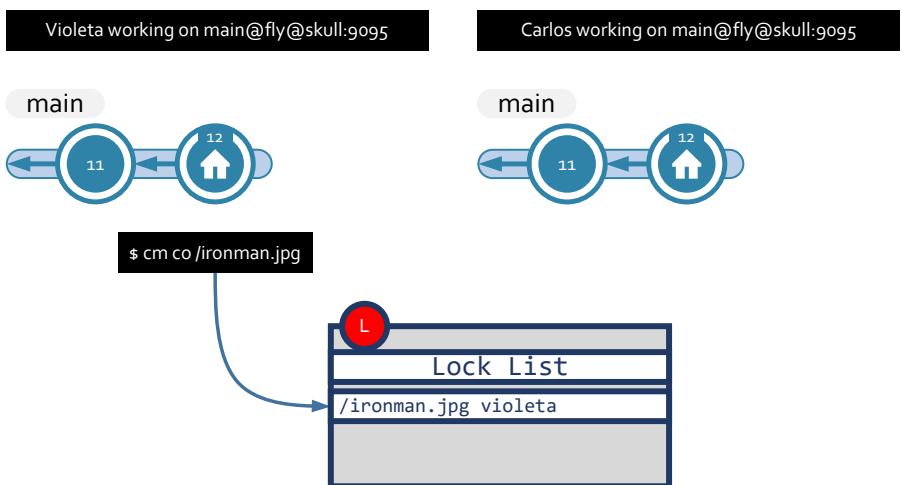
Lock to prevent concurrent changes

Plastic can be configured to lock certain files and prevent concurrent changes on them (check [this](#) [<https://www.plasticscm.com/download/help/locking>] for more info).

I'm going to explain how it works and what the team needs to do to take advantage of this feature.

First, the sysadmin needs to configure locking on the server for certain file extensions. In my case, I'm going to consider all `.jpg` files are locked on checkout.

Then, let's consider the following starting point:



Violeta and Carlos are working on the same project. Violeta decides to checkout the file `ironman.jpg`. In the figure, the checkout is done using the command line but of course the same operation can be done from a GUI.

Checkout is an operation to tell Plastic "I'm going to modify this file". Normally, it is not needed because you can simply modify the file without telling Plastic about it, and it will find out later.

But, checkout works differently for files configured to be locked. When you checkout `ironman.jpg`, since it is configured to be locked, a new entry is created in the server lock list meaning that `ironman.jpg` can't be modified concurrently.

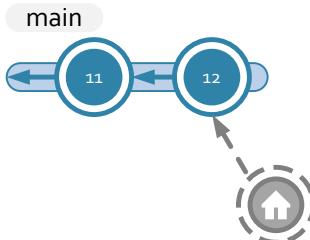


To take advantage of locks, team members need to checkout the files. If they forget about it, then locks won't be obtained or requested, and there won't be any way to prevent unmergeable files from being modified concurrently.

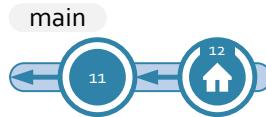
In our example, Violeta correctly checked out `ironman.jpg` and the file was added to the server lock list. I just added the name of the file to the list in the figure above, but in reality, info about the itemid is also added to properly identify the locked files.

Suppose that now Carlos tries to checkout `ironman.jpg` too:

Violeta working on main@fly@skull:9095



Carlos working on main@fly@skull:9095



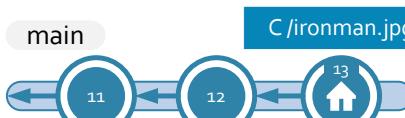
\$ cm co /ironman.jpg
Error => ironman.jpg is already locked

CO /ironman.jpg

The checkout operation fails because the file is already locked by Violeta in a different workspace. This is how we can prevent unmergeable files from being wrongly modified concurrently.

Later Violeta finishes her work and checkins the changes. A new changeset 13 is created on `main` and the entry in the lock list is now cleared.

Violeta working on main@fly@skull:9095

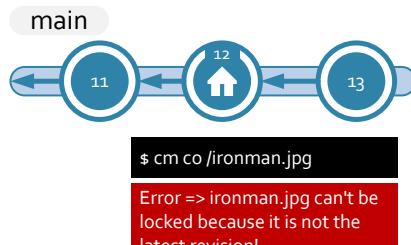


\$ cm ci /ironman.jpg



Now, Carlos is free to lock `ironman.jpg` to perform changes, correct? Not quite yet:

Carlos working on main@fly@skull:9095

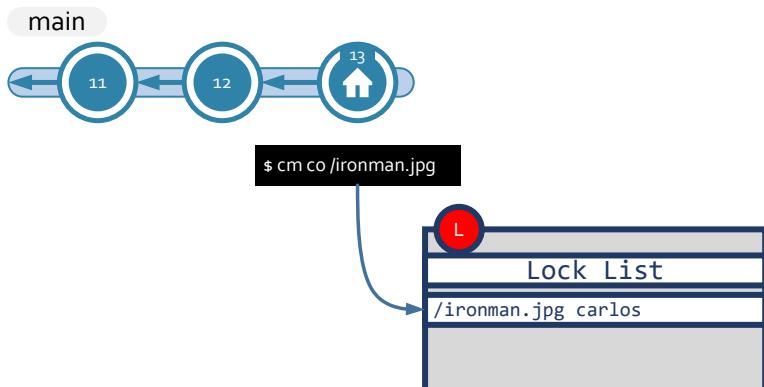


Carlos is still on changeset 12 in his workspace, so he can't lock the file because, while there aren't locks for `ironman.jpg` held at this point, his revision is not the latest.

Plastic ensures you can only lock the latest version of a file otherwise concurrent modification wouldn't be prevented either.

Then, Carlos decides to update his workspace to changeset 13 and he is finally able to lock the file.

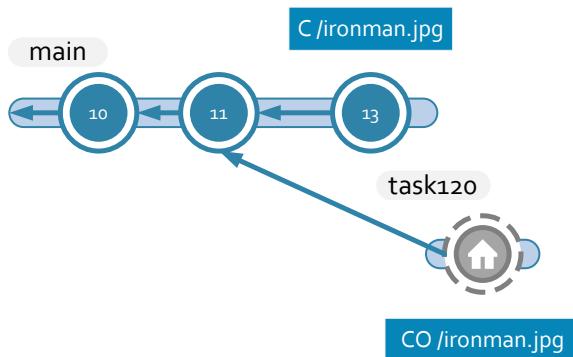
Carlos working on main@fly@skull:9095



This way the changes in the `jpg` file will be serialized since nobody will be able to make concurrent modifications.

Locks only work in single branch

This is a strong remark: The Plastic locking system only works well if you work on a single branch. We didn't implement the ability to coordinate locks among branches. You can easily force the situation where you held a lock on a branch, so nobody else can modify the file, then checkin. At this point, someone else could lock the file on a different branch. But this won't obviously prevent concurrent modification.



In the figure, someone created changeset 13 after locking `ironman.jpg` on `main`.

Later someone creates a branch starting from changeset 11 and locks the file (checkout). His version of `ironman.jpg` is the latest on his branch, so he can perform the lock, but obviously we are not preventing merges to happen since this is in fact a concurrent modification.

So, keep in mind that locking is very useful to deal with unmergeable files, but you need to restrict its use to single branch operation.

A glimpse to the future: travelling locks

At the time of this writing, we are considering a great improvement in the locking system: travelling locks.

What if you could lock a file on a branch and specify that it shouldn't be unlocked until it reaches `main` on a given server?

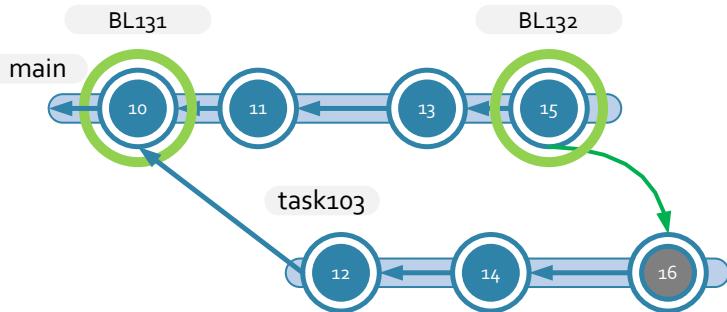
This will enable to safely modify unmergeable files in a branch, even make several checkins on the branch. It will even allow working on different repos and still hold the lock until the revision reaches main on the central server. At this point the merge will be freed.

The solution is not simple since there are many possible lock propagation scenarios (what if you branch off from where you acquired the lock and then modify the file again? This would lead to endless locks that would never be freed if they don't all end up being merged to main, for instance), but it is part of our idea to simplify workflows for teams where binaries and assets are part of their daily workflows.

Plastic rebase vs. Git rebase

We often use the word rebase to refer to a merge from `main` (or a release branch) to a branch of lower hierarchy.

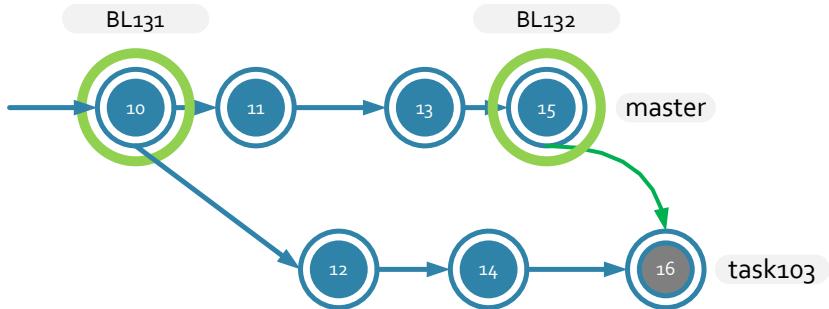
A typical rebase in Plastic is something as follows:



You can see how the branch `task103` started from `BL131` (it was "its base") and then later we wanted to update the branch with the latest changes in `BL132` (changed the base, so we call it "rebase").

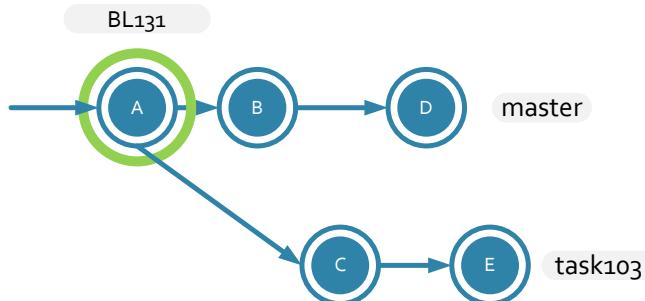
But, if you are coming from a Git background, or simply want to become a master in version control ☺, then it is worth pointing that the term rebase has a completely different meaning in Git.

Of course, it is possible to do something as follows in Git:

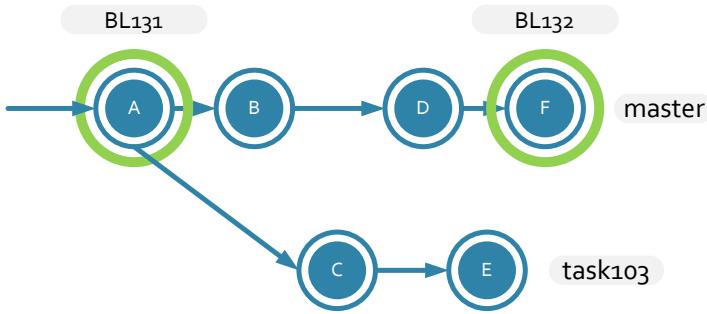


Which would be the exact equivalent to the operation made above in Plastic.

But, let's rewind a little bit to better explain the scenario in Git. You start working on a branch and perform a few commits on it. And meanwhile, the master branch that you branched from also evolved.

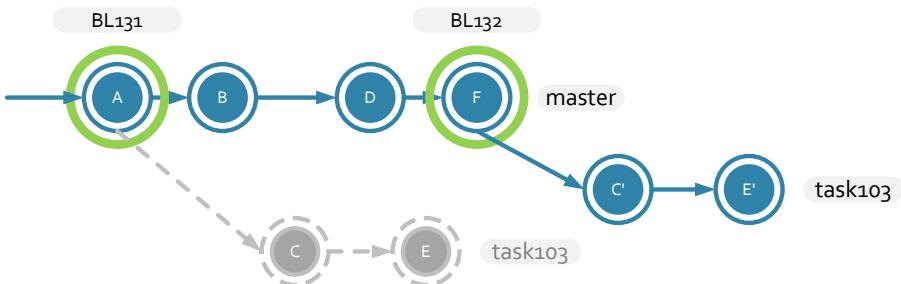


Then, at a certain point in time a new commit is tagged in `master` and you want to bring those changes to `task103`. One option would be to simply merge down as we did above.



But, in Git you can do a "rebase" which is a history rewriting operation. Basically, a rebase creates new commits **C'** and **E'** which resemble the changes made in the original **C** and **E** but starting from **F**. Actual file merges can happen during the rebase.

The old original commits **C** and **E** are left in the repo but they are now inaccessible and will be removed in the next repo garbage collection operation.



It is totally discouraged to do a rebase if you already pushed your branch **task103** to a public repo because it will lead to a number of issues since you later changed the history locally, but propagating the change to other repos is not that simple. Rebase in Git is an operation meant to be run only locally and prior to pushing the commits to a shared location.

So far, (I'd never say never), we haven't implemented history rewriting ourselves since we prefer to think of repos as what really happened instead of a rearranged history. Git made rebases super popular so whether we end up implementing something similar will only depend on user demand.

There is one thing to keep in mind, though: one of the main reasons to do rebases in Git is to simplify understanding the changes, because linear history is easier to diff than when changes come from merges. Keep in mind that:

- Diffing a branch in Plastic is straightforward because Plastic knows where the branch starts and ends. Knowing where the branch starts and ends is commonly impossible to determine in Git due to fast-forward merges and the extended practice of deleting branches.
- Git doesn't provide built-in diff tools. Yes, there are many third-party tools that do side-by-side diff, but the default package just comes with unified diffs suitable for the command line. Plastic, in contrast, excels in diff tools (it even comes with built-in SemanticDiff) which makes explaining differences easier. My point here is that rebase is often needed in Git to workaround with history rewriting what a proper diff tool can do. In Plastic, a diff will tell you which files and lines come from a merge and which don't. Check the following screenshot to see a diff of a branch involving a merge:

The screenshot shows a code comparison interface in Plastic SCM. The top navigation bar includes 'Item: 1/6', 'Statistics', 'Analyze differences', 'Analyze refactors', 'Find in files...', and a 'Filter' field. Below this is a table of file changes:

Path	Created by	Date modified	Repository	Viewed
Merged: 1 item └ /code/server/FileSystem.cs	pablo	10/15/2015 16:15:38	quake@local	viewed
Added: 2 items └ /code/server/SeriesController.cs	pablo	10/28/2015 16:00:27	quake@local	viewed
└ /propulsion/chiquito.txt	pablo	11/16/2015 10:28:34	quake@local	
Moved: 2 items └ /code/server-renamed to /code/server	pablo	10/28/2015 16:00:27	quake@local	
└ /code/splines/FileSystem-rename.cs to /code/server/FileSystem.cs	pablo	10/15/2015 16:15:38	quake@local	
Merged from cs619@br/main@quake@local: 1 item └ /code/server/FileSystem.cs	pablo	10/15/2015 16:15:38	quake@local	

The main area displays the semantic outline and visual diff for the file /code/server/FileSystem.cs. The semantic outline on the left shows moved and changed methods. The visual diff on the right shows added and deleted code blocks.

Learn more about Git vs. Plastic diffing abilities reading [this blogpost](#) [<https://www.plasticscm.com/download/help/plasticrebasevsgitrebase>].

WORKSPACES

The workspace is a central part in the interaction with the version control. It is the directory that you use to interact with the version control, where you download the files and make the required changes for each checkin.

In this chapter, we're going to explore the two different types of workspaces in Plastic: full workspaces and partial workspaces. We're going to delve into how workspaces are configured, how they work and how they shape the way in which Plastic works.

Full workspaces vs. partial workspaces

Plastic SCM supports two different types of workspaces:

Full workspaces

Also known as "standard workspaces", "traditional workspaces", "Plastic workspaces", and "classic workspaces". "Full workspace" is the way to standardize how to refer to them. Full workspaces are in sync with a changeset at any given point in time. It means the contents you have on disk resemble a changeset on the server plus any local changes you have made. Full workspaces are suitable for running merges and normally the ones chosen by developers.

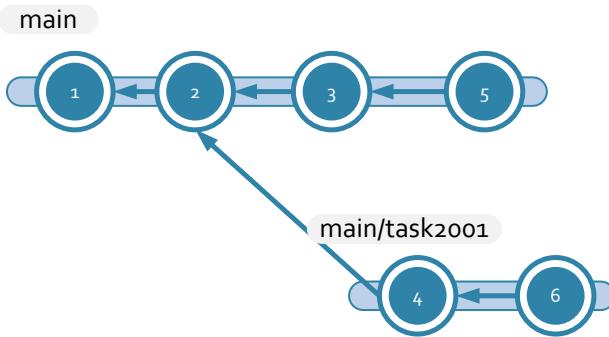
Partial workspaces

Also known as "Gluon workspaces". We introduced them together with Gluon, the UI and workflow for non-coders, typically artists in game development. They were built around three key requirements: don't download the entire tree of a changeset, merge is never required, and allow to work in a per-file basis (being able to checkin even if part of the workspace is out of date with head).

If you didn't understand part of the previous definitions, don't worry; the goal of this chapter is to ensure you master both types. Keep reading ☺

What is a workspace

Imagine you have a repository that looks like the following:



And then, you create a workspace to work on it. A new workspace is initially empty until you run an update, which is the operation responsible for downloading the files and directories from the repo to your disk.

In our example, imagine we run the following combination of commands:

```
cm workspace create wk00 c:\users\pablo\wkspaces\wk00 --repository=quake@skull:8084
cd c:\users\pablo\wkspaces\wk00
cm switch main/task2001
```

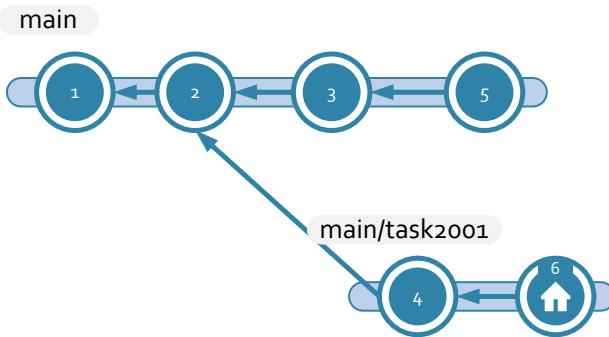


I'm using Windows paths, but it will work exactly the same using Linux/macOS ones. And I'm using the command line, but you can achieve exactly the same behavior from the GUI.

The first command, `cm workspace create`, creates a new workspace and sets `quake@skull:8084` as the repo to work with. By default, the `main` branch is where new workspaces point to.

The second command, `cm switch`, switches the workspace to a given branch, `main/task2001` in our case, and runs an update.

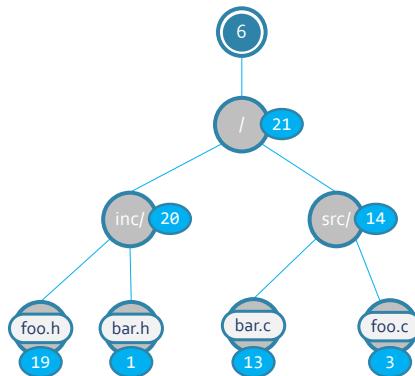
If you check your Branch Explorer at this point it will look like this:



The home icon in the Branch Explorer highlights where the workspace points to. In our case, it points to latest on `task2001`, or the changeset number `6`.

What happened when you run the switch?

Remember the chapter "Merging trees" where we went in great detail through how every single changeset points to a given tree. Let's consider now that the tree in changeset `6` is something as follows:



The result of the `switch` (which is actually like an `update`) will be the following directory structure on your workspace: (in my case it was `c:\users\pablo\wkspaces\wk00`)

```

/
/inc/
/inc/foo.h
/inc/bar.h
/src/
/src/bar.c
/src/foo.c

```

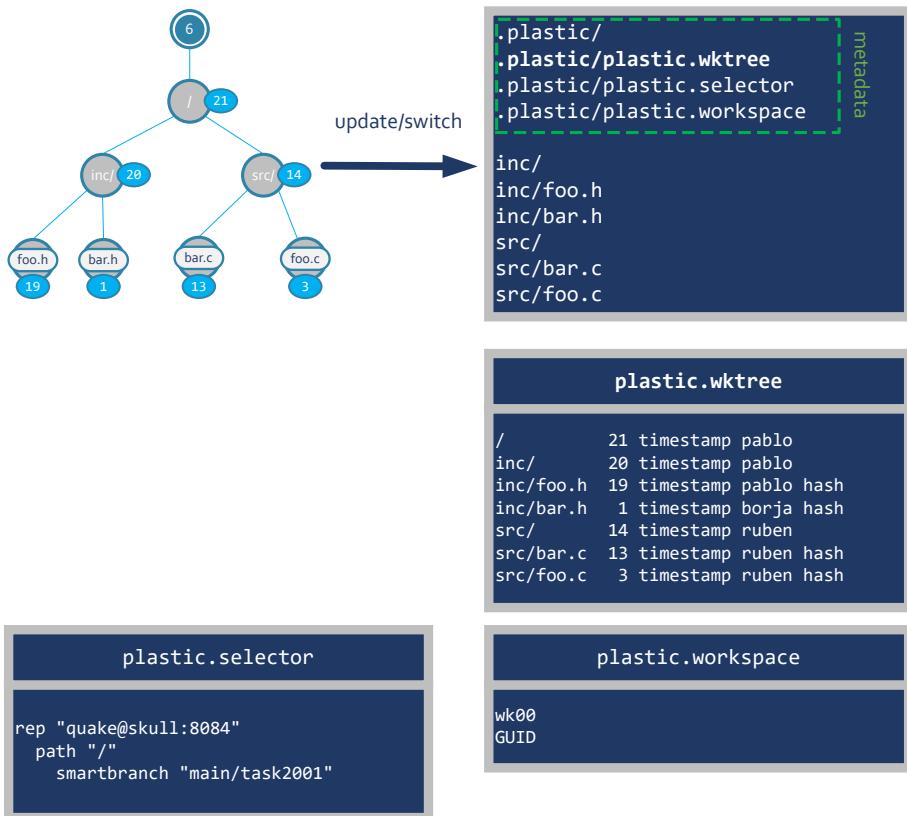
Where the files will have exactly the contents of the corresponding revisions on the repository.

A full workspace is just a copy on your disk of the actual structure of a given changeset. In the case of a partial workspace this definition will vary a little bit, since the copy won't have to match just a single changeset.

Metadata of a workspace

Let's rewind a little bit and let's see again what happens when you download data to an empty workspace.

Together with the actual files being downloaded for the repo, Plastic creates (or updates) three key files inside the `.plastic` hidden directory in the root of the workspace: `plastic.wktree`, `plastic.selector` and `plastic.workspace`.



- **plastic.selector** simply contains information about the repository and the current branch. The name "selector" is kept for historical reasons. In pre-4.0 versions, it contained rules to select which revisions to load, but this is all gone now.
- **plastic.workspace** contains the workspace name and a GUID to unequivocally identify it.
- **plastic.wktree** is where the really important metadata is stored. It contains metadata about the revisions downloaded from the repo that will be later used to track changes in the workspace. The metadata contains the name of the file or directory, the revision number, a timestamp of when it was written to disk, a hash of its content downloaded from the repo, and then extra info to display in GUIs and command line like the author and a few more.



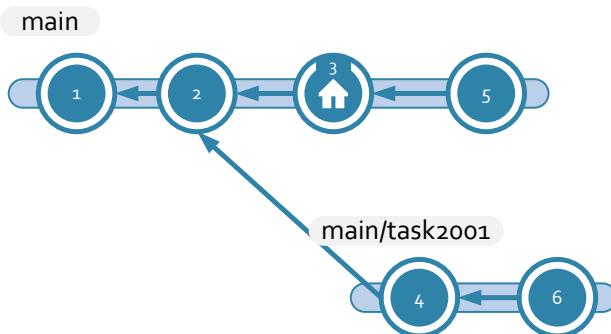
File timestamp on plastic.wktree: stores the date when the file was written to the workspace, not when it was created in the repo. This way workspaces are build-system friendly; when you get a different version of a file, its date on disk will be newer, even if the revision is older, making build systems trigger a build because there are changes (think Ant, Makefile, etc).

Update and switch operations

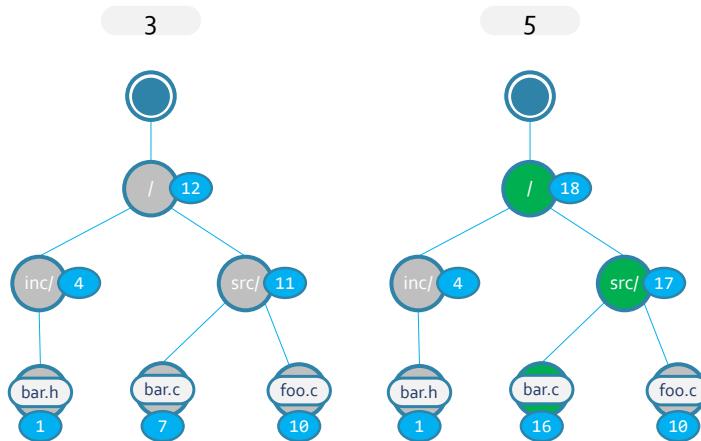
In the first example, I ran a `switch` which is the command used to point the workspace to a different changeset or branch. The update operation is an equivalent to switch that simply says "switch to whatever is latest in my current configuration".

Update

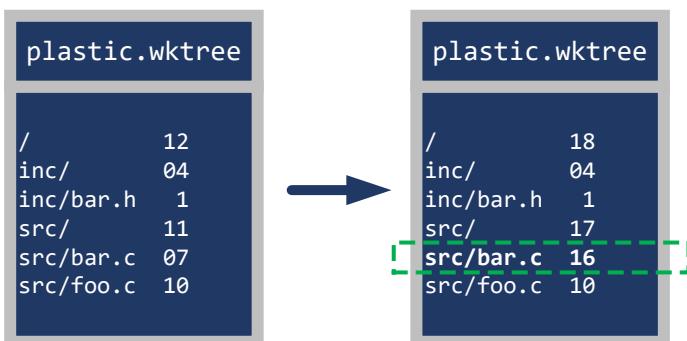
Let's continue with the same sample repo, but suppose my workspace was located on changeset 3 instead and then we decide to run an update.



The structure of the trees of the changesets 3 and 5 are as follows:



So, when you decide to run an update, Plastic will realize it is in changeset 3 and it has to jump to changeset 5. It knows it just has to download a new version for `bar.c` replacing the current one. At the same time, it downloads new data to update `/src/bar.c`, it will update `plastic.wktree` accordingly.



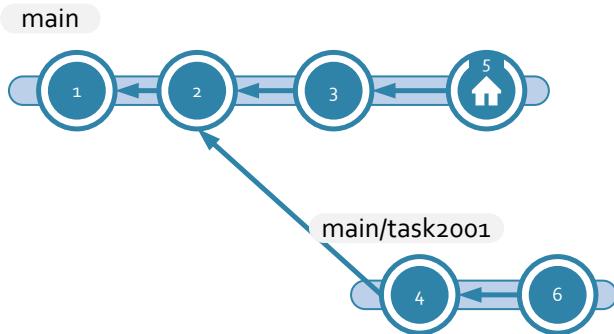
Update simply moves the workspace to its latest configuration. So, if your workspace is set to work on

main and you run an update, Plastic will locate whatever the head of the branch is and will try to jump to it updating both data and metadata in the workspace.

Switch

A switch is an update that instead of going to "latest", jumps to a different configuration, typically a different branch or changeset.

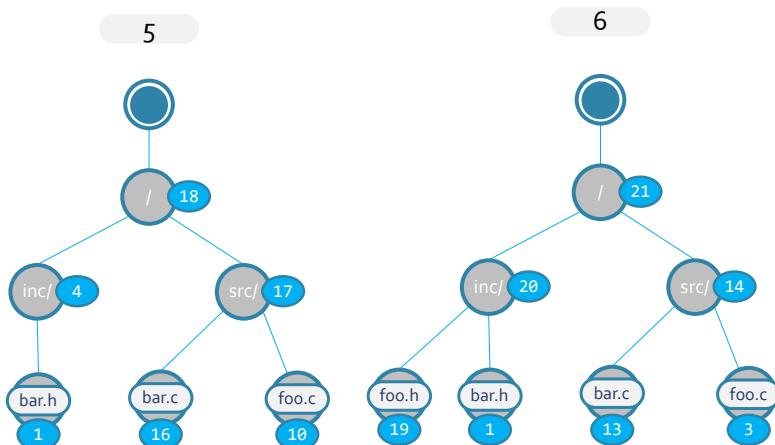
Let's start from where we left in the previous section:



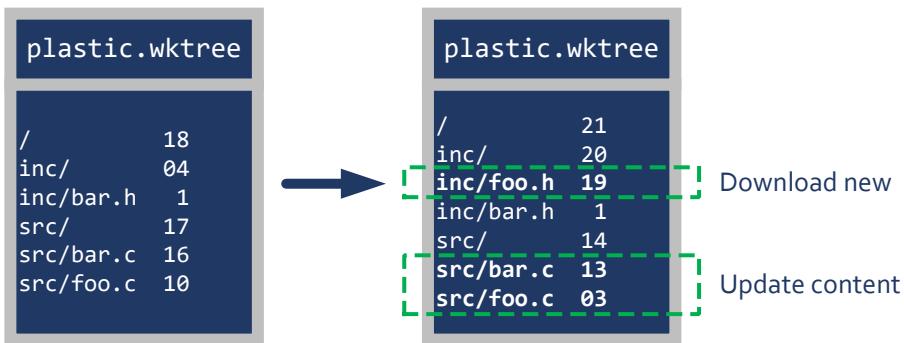
And let's run a switch to **main/task2001**:

```
cm switch main/task2001
```

Plastic will check the trees of the changesets **5** and **6**:



And then figure out what needs to be downloaded to convert your working copy into changeset **6**.



In this case, a new file `inc/foo.h` will be downloaded from the repo (it didn't exist on the workspace before) and the contents of both `src/bar.c` and `src/foo.c` will be updated to different revisions.

Update to a different repo

It is possible to update a workspace to a different repository. In Plastic, workspaces are not tied to a given repo, and you can reuse a workspace to work with a different repository at any time.

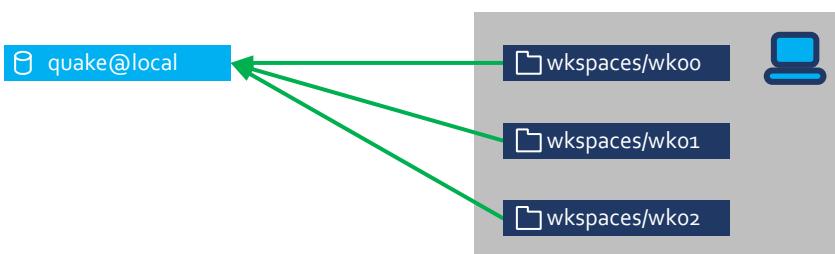
In fact, the update and switch operations will try to reuse as much data as possible to reduce the traffic with the repo (which might be a distant server). What I mean by this is that, even if the revids don't match, Plastic will try to reuse files based on hashes. So, if you switch your workspace from `main/task2001@quake@skull:8084` to `main/task2001@quakereplica@localhost:8084` where `quake` and `quakereplica` are actually replicas of the same repo, Plastic will try to reuse all the files on disk and if changeset 6 exists on both, no new file data will be downloaded.

The only practical reason why you wouldn't switch a workspace, though, is size. I mean, suppose your workspace is 3GB and you have local files that add another 2.5GB. If this workspace points to `quake@skull:8084` and you plan to continue working on this repo, there is no good reason to switch to `doom@skull:8084` and have to download another 5GB and discard the 2.5GB of local private files.

One repo many workspaces

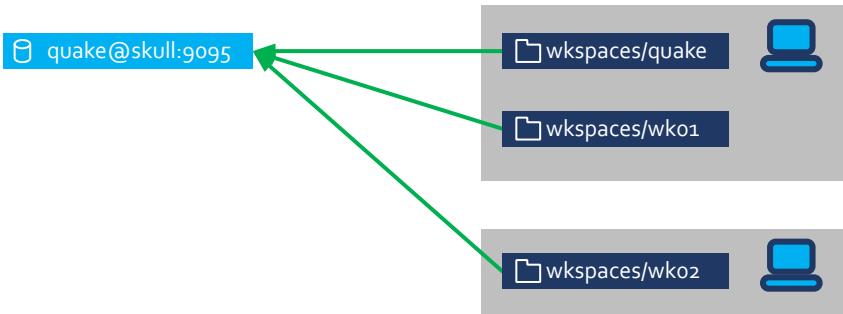
Although it is already clear that workspaces in Plastic can be used to switch to different repos, it is important to highlight that more than one workspace can simultaneously work connected to the same repo.

You can have many workspaces pointing to the same local repo:



And there can be different workspaces on different machines directly working with a repo on a remote

server:



This might sound rather obvious to most of you, but if you are coming from Git, you'll be used to a one-to-one relationship between working copy and repo, since this is the Git architecture. In fact, there is no possible way to have working copies directly connected to a remote repository since a local intermediate repo is always required.

Tuning the update operation in full workspaces

When you run an update or switch operation, there are a few configuration files that affect how the files are actually written to disk.

These files can be private to the workspace, can be part of the repository (checkedin to the root dir of the repo), or can be global files. You can learn more about the specifics of how to setup these files in the [Administrator's Guide, Configuration Files section](#) [<https://www.plasticscm.com/download/help/configfiles>].

This section won't get into the details about the actual formats, but it will explain what they do in depth.



Cloaked

`cloaked.conf` defines the paths to cloak in the workspace. The file is defined as a set of user-defined regular expressions. The files that match any of the rules will be cloaked.

Cloaked works in two different ways:

1. Suppose you already have `/src/foo.c` in your workspace and then you decide to cloak it. This means the file will be ignored by the update and hence never be updated even if new versions exist. It is useful if you want to avoid downloading certain big files or a huge tree, because you can continue working with what you have and want to save time in updates.
2. Suppose your workspace is empty and a `cloaked.conf` that matches `/src/foo.c` exists in the repo, or

you simply create one on your workspace. This means `/src/foo.c` won't be downloaded. It is useful to prune certain parts of the tree, normally due to size issues.



The merge operation might ignore the cloaked files when the file needs to be downloaded to resolve a merge.

You can [read the details of how to setup `cloaked.conf`](#) and check the [reference guide](#) [<https://www.plasticscm.com/download/help/addtocloakedlist>] to learn how to configure it from the GUI.

Readonly and writable

By default, Plastic sets all files on the workspace as writable. It is the default behavior since most developers simply want to edit a file and save without having to deal with overriding the read-only attribute.

But, sometimes it is useful to set files as read-only so that users see an operating system warning asking them to override the read-only setting when they hit save in their applications. This is commonly used by team members working on assets that need to be locked, as a reminder to checkout the files prior to saving them.

Plastic provides a global setting to decide whether update should set files as writable (the default) or read-only.

The `writable.conf` and `readonly.conf` config files are useful to create exceptions to the global setting.

In a default "everything is writable" configuration, `readonly.conf` will define which extensions or file patterns need to be set as read-only on the workspace.

And `writable.conf` does exactly the opposite.

You can [read the details of how to setup `writable.conf` and `readonly.conf`](#) and check the [reference guide](#) [<https://www.plasticscm.com/download/help/setfilesasreadonlyorwritable>] to learn how to configure this setting from the GUI.

Tune EOLs

I have to admit this is one of my least favorite settings and we only added it to ease the transition of some teams to Plastic from their previous version controls.

The file `eolconversion.conf` defines which files need to convert end of lines during the update and reverted back to their original during checkin.

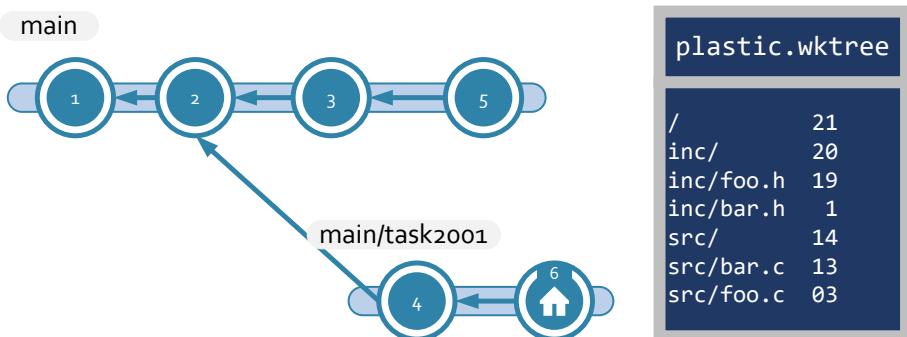
I don't like this option because otherwise Plastic never touches the file content during update or checkin, preserving exactly what the developer created. Converting EOLs introduces a performance impact and breaking the "we preserve what you wrote" rule.

Nevertheless, some teams prefer to see Windows-style EOLs on Windows, and Linux-style EOLs on Linux and macOS. Nowadays, almost all editors I'm aware of are capable of handling EOLs transparently, but still some teams and some tools prefer that the version control normalizes the end of lines.

Learn how to exactly [specify `eolconversion.conf`](#) [<https://www.plasticscm.com/download/help/eolconversion>].

Finding changes

Let's go back to where we left our sample workspace in the previous example, having just switched to `main/task2001`:



The `plastic.wktree` contains the metadata of the files "controlled" by the workspace, the ones that resemble what is in the repo.

Looking for changes

Each time you run a `cm status` on the command line or go to "pending changes" or "checkin changes" in the GUIs (we use the two names interchangeably), Plastic walks the workspace to find if there are any changes.

In fact, it groups the changes in the following sections:

- Files and directories to add.
- Changed files.
- Deleted files and directories.
- Moved/renamed files and directories.

Check the following figure, based on the scenario above where the workspace points to changeset 6:

- It finds that `inc/bar.h` exists in the metadata, but no longer exists on disk, so it is considered a "locally deleted".
- The timestamp of `src/foo.c` on disk is newer than the one stored in the metadata, so the file is considered as changed. There is an option in the Plastic settings to do a double check: if the timestamp changes calculate the hash of the file on disk and compare it with the one stored in the metadata to mark the file as changed. Just checking the timestamp is much faster, but there are tools that tend to rewrite files even if they didn't change, so enforcing the hash check is sometimes very useful.
- `src/new.c` exists on disk but didn't exist on the workspace tree, so it is marked as a private file that is a candidate to be added to the repo in the next checkin. More about private files in "[Private files and ignore.conf](#)".

plastic.wktree				Actual workspace content	
Name	RevId	Written	Hash		
/	21			/	
inc/	20			inc/	
inc/foo.h	19	2018/12/24 13:31	hash	inc/foo.h	2018/12/24 13:31
inc/bar.h	1	2018/12/24 13:31	hash		D inc/bar.h
src/	14			src/	
src/bar.c	13	2018/12/24 13:31	hash	src/bar.c	2018/12/24 13:31
src/foo.c	03	2018/12/24 13:31	hash	src/foo.c	2018/12/24 13:42
				src/new.c	2018/12/24 13:45
					C src/foo.c A src/new.c

I didn't cover how moved and renames are detected since it has its own section.

Now, you have a pretty good understanding of how changes are detected in a workspace.

Private files and ignore.conf

Any file or directory you add to the workspace will be considered as private. Private means it is just local to the workspace and won't be checked in and changes won't be tracked.

Each time you ask Plastic to find changes, the local files will show up marked as private.

All these privates are suggested by the GUIs to be added to source control.

Very often that's exactly what you want to achieve, because you just added a new file and want to add it to version control.

But there are many files, like local configs or intermediate build files (obj, bin, etc) that you simply want to ignore so Plastic doesn't show them again and again.

All you have to do is to add them to `ignore.conf` and Plastic will, well, ignore them 😊

Although adding files to the `ignore.conf` list is very easy from GUIs or simply editing the file, you can [read the details of how to setup `ignore.conf`](#).

Detecting moves and renames

In the chapter about merges, you already saw how good Plastic is dealing with moved files and directories and how important it is to help refactoring the code base.

But, before a move is checked in to the repo, first it has to be detected on the workspace. This is exactly what I'm going to explain now.

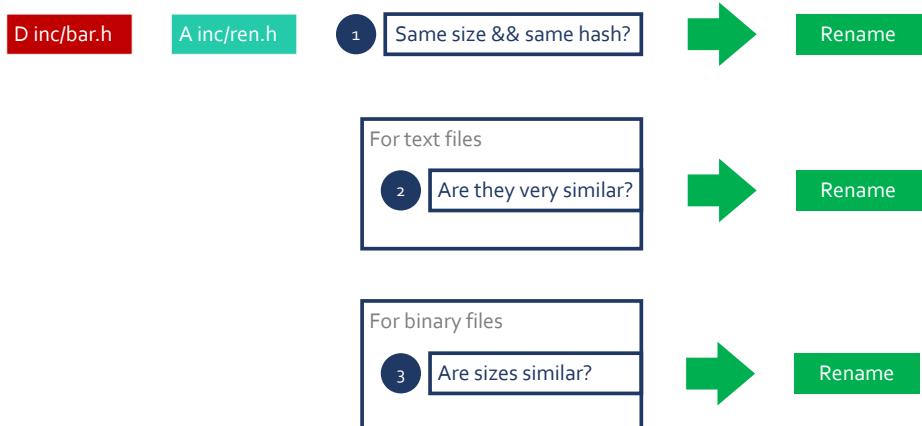
Let's start again with changeset 6 in the examples above and consider the following local changes:

plastic.wktree				Actual workspace content			
Name	RevId	Written	Hash	Name	Written		
/	21			/			
inc/	20			inc/			
inc/foo.h	19	2018/12/24 13:31	hash	inc/foo.h	2018/12/24 13:31		
inc/bar.h	1	2018/12/24 13:31	hash			D inc/bar.h	
src/	14					A inc/ren.h	
src/bar.c	13	2018/12/24 13:31	hash	inc/ren.h	2018/12/24 13:45		
src/foo.c	03	2018/12/24 13:31	hash	src/			
				src/bar.c	2018/12/24 13:31		
				src/foo.c	2018/12/24 13:31		

The file `inc/bar.h` is no longer on disk and a new one `inc/ren.h` appeared as new.

If move detection is enabled (it is by default, but it can be disabled), Plastic tries to match added/deleted pairs to try to find moves.

The following figure gives an overview of the algorithm:



- If `bar.h` and `ren.h` have exactly the same size, then Plastic checks their hashes. If they match (`bar.h` taken from metadata and `ren.h` calculated from disk) then it means it found a rename (or a move, which is exactly the same as a rename but happens between different directories).
- If files are considered to be text files, then our similarity algorithm will try to find how similar the files are. Since the hashes and sizes don't match, it means the file was modified after being renamed. So, if the similarity % is higher than a given configurable threshold, the pair will be considered as a rename/move.
- Finally, if the files are binaries, the similarity algorithm won't work, so what Plastic does is compare their sizes. If the sizes are very similar, then binaries are marked as moved. This is a best guess because it is not that simple to figure out that a PNG was renamed if it was later modified, but we try to refine the heuristic to avoid false positives.

The move/rename detection also works for directories. Let's see it with an example: suppose we rename the `inc/` directory to `incl/`. What the change detection algorithm will see is something as follows:

plastic.wktree				
Name	RevId	Written	Hash	
/	21			
inc/	20			
inc/foo.h	19	2018/12/24 13:31	hash	
inc/bar.h	1	2018/12/24 13:31	hash	
src/	14			
src/bar.c	13	2018/12/24 13:31	hash	
src/foo.c	03	2018/12/24 13:31	hash	

Actual workspace content				
Name	Written			
/				
incl/				
incl/foo.h	2018/12/24 13:31			
incl/ren.h	2018/12/24 13:31			
src/				
src/bar.c	2018/12/24 13:31			
src/foo.c	2018/12/24 13:31			

It finds that `inc/` and its children are no longer on disk, and a new tree `incl/` has been added.

Then, the added/deleted pairs will try to be matched. In this case, the algorithm finds that `inc/foo.h` and `incl/foo.h` have the same size and content, so they must be the same file. And the same happens for `ren.h`.

But instead of showing a pair of moves, the Plastic algorithm goes one step further: it tries to match `inc/` and `incl/`. Since its contents are exactly the same, it detects a rename of inc to incl and cleans up the individual file moves.

In case the files were modified, deleted or moved after the directory rename, Plastic will apply a similarity percentage to the `inc/` directory structure. If the structures are similar up to a given %, then the directory will be considered as a move/rename instead of added/deleted.

We are very proud of move detection and we gave this feature a big name: "transparent scm" or "transparent version control" because you can simply work in your workspace, make any changes, and trust that Plastic will detect what you have done.

We use a number of extra techniques to detect moves, but I'll cover that in its own section.

Hiding changes with `hidden_changes.conf`

Just like you sometimes need to ignore private files with `ignore.conf`, there are scenarios where you don't want Plastic to detect changes in certain files because you don't want to mistakenly include them in the next checkin. This can happen if you simply need to adjust config files for a debug session, or alter some source code for whatever reason, but you don't want to checkin those temporary changes.

`hidden_changes.conf` helps achieve that. You can [read the details of how to setup `hidden_changes.conf`](#) and check the [reference guide](#) [<https://www.plasticscm.com/download/help/addtohiddenlist>] to learn how to configure it from the GUI.

Controlled changes - checkouts

So far, we were doing changes in the workspace and let Plastic detect what we did.

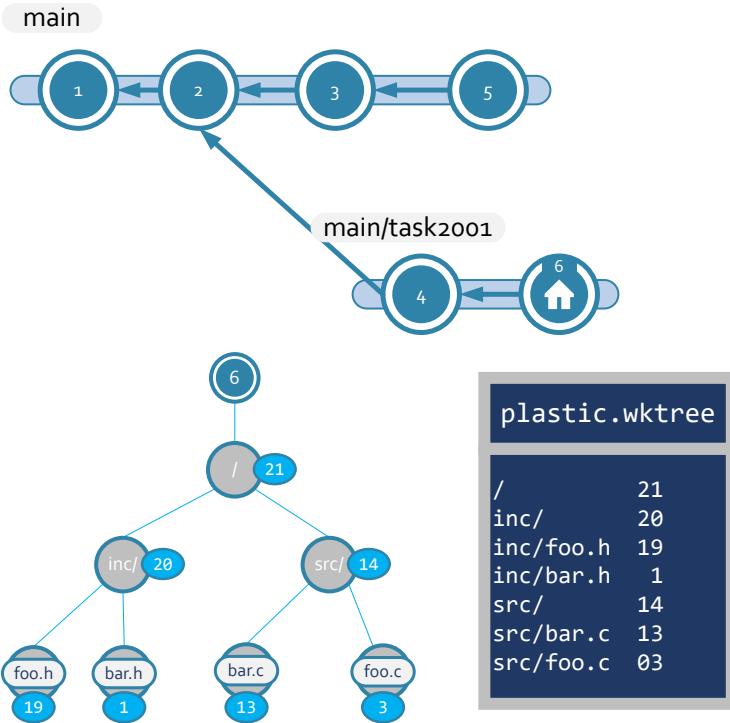
But there is another alternative: telling Plastic exactly what you changed, moved or deleted so it doesn't have to "guess".

We call these "controlled changes" and "checkouts" are a subset of them, but normally we use "checkouts" for all changes in the workspace that are "controlled" by Plastic because you notified about them.

Notifying added, deleted and moved

Suppose you want to heavily refactor your project and you can simply expect Plastic to correctly guess what you did after the fact based on similarity algorithms. What is the alternative?

Let's start again from the tree in changeset 6:

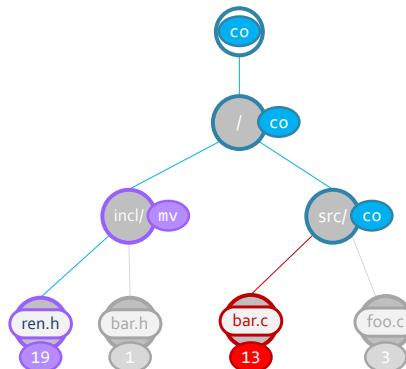


And let's perform the following actions:

```
cm mv inc incl  
cm mv incl/foo.h incl/ren.h  
cm rm src/foo.c  
echo foo > readme.txt  
cm add readme.txt
```

Now, let's check the metadata of the workspace:

plastic.wktree				plastic.changes			
/	21			added /readme.txt			
readme.txt	C0	added			mv /inc /incl		
incl/	20		checkout/moved			mv /incl/foo.h /incl/ren.h	
incl/ren.h	19		checkout/moved			rm /src/foo.c	
incl/bar.h	1						
src/	14						
src/bar.c	13						
src/foo.c	03	deleted					



As you can see in the figure:

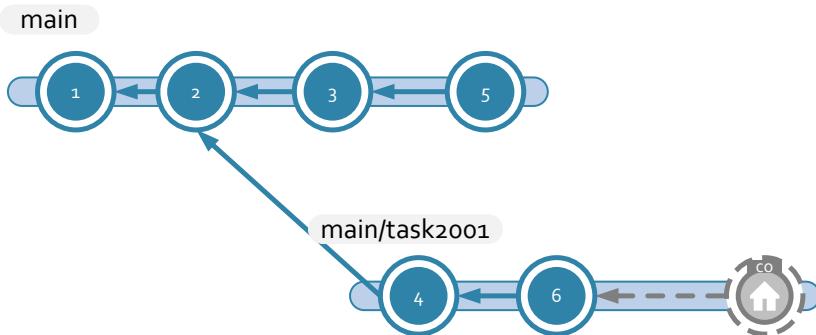
- `plastic.wktree` now contains information about the actual changes. It knows `readme.txt` was added, that `/incl` was renamed and also `/incl/ren.h` was renamed. It also knows that `src/foo.c` was deleted.
- `plastic.changes` is a new file that is now present in the `.plastic` metadata directory. It complements `plastic.wktree` with information about the actual changes. While `plastic.wktree` just knows that `/incl` was renamed, `plastic.changes` contains the data of the actual rename, the source of the move, and so on. In fact, `plastic.changes` doesn't store the info as a list of operations but more like a tree (also displayed in the figure). This tree is actually a subtree of the workspace metadata that is ready to be sent to the server and contains extra info for the nodes that were moved to univocally identify the sources of the moves/renames. It is a subtree decorated with extra info telling the operation that was applied to create it.

Note that I marked the tree with a "CO" instead of a number. CO stands for checkout. So what `plastic.changes` contains is what we call "the checkout tree", which is a tree that contains the local changes controlled by Plastic and ready to be checked in.

There is something very important to note: when you tell Plastic to do a move, add, delete, Plastic doesn't have to guess these operations anymore. Unlike what was described in the "["Finding changes"](#)" section, these are controlled changes, Plastic knows about them because you told it you did them, so there is no guessing or heuristics involved.

One of the key advantages of this alternative is to handle moves and renames; there is no guessing involved, so you can perform extremely complex refactors with "moves" and Plastic will track exactly what you did.

After performing these changes, if you go to the Branch Explorer, you'll see something as follows:



You see how there is a new "checkout changeset" and the home icon is on it, meaning your workspace is not synced anymore to changeset 6 but 6 plus some controlled changes on your workspace.

If you make not-controlled changes (just make the moves and deletes and let Plastic detect them but you don't tell it what you did), then there won't be a checkout changeset in your Branch Explorer.

How to move files and directories from GUIs

We have seen the `cm mv` command to move files and directories from the command line. But, you can achieve the same from any of the GUIs going to the "Workspace Explorer" view (previously known as "items view") and doing CTRL-X on the source, and then pasting the destination with CTRL-V (or the equivalent command key on macOS).

For renames, it is simpler; just show the context menu of the file or directory to rename and you'll find a "rename" option. Of course, you can use the particular rename shortcut on your operating system.

Notifying changes

You can tell Plastic you are going to modify a file by using the checkout action both from GUIs (right click a file from "Workspace Explorer" and select "checkout") and command line (`cm co` command).

Let's continue with the example above. You can perform:

```
cm co src/bar.c
```

And Plastic will mark the file as checkedout both in `plastic.wktree` and `plastic.changes`.

This is what we strictly call a checkout, although, as we saw in the previous section, we extend the name checkout for controlled deletes, moved and added files.

When to use controlled changes

Very simply, to control changes:

- If you need to lock files, you must checkout them first. Check "[Locking: avoiding merges for](#)

"[unmergeable files](#)" for more info. Never forget to checkout before making any changes to files you need to lock, or there will be a risk of losing your local changes if someone else locked the files and modified them first.

- If you want to perform a very complex refactor, chances are you are better served by `cm mv` (or GUI equivalent) than letting the heuristics guess what you did after the fact.
- If you have a huge workspace (bigger than 700k files) and the performance of detecting changes is not fast enough (it takes more than 4-5 seconds, which starts to be really annoying) then you probably are better served by checkouts. To achieve this, you have to disable finding local changes and stick to checkouts. There is an option in "Pending Changes" (or "Checkin Changes") in all GUIs to do this: by disabling it you make Plastic just show the checkouts, and ignore any changes done locally, so it relies on you telling it what you did.

What to find What to show Move detection

Show checkouts
Shows files you explicitly checkedout in Plastic. Also files you moved using Plastic GUIs or command line. And finally files involved in a merge in progress.
Checkouts display super-fast because they are on a list; no need to walk the directory structure to find them.
Tip: With a workspace larger than 800k files, it may make sense to use only checkouts for faster operation. Checkouts are also needed if you use file locks.

Find changed files in the workspace
Walks the workspace directory structure to find changed files (timestamp changed).

Check the content to mark files as changed, not only timestamp
When the timestamp of a file is modified, calculate the hash with the previous before setting it as changed. Useful with applications that save files without performing any changes. Warning: It can make "pending changes" slower because it requires reading and hashing files.

- You'll be using controlled changes transparently if you enable your plugin for Visual Studio, Eclipse, IntelliJ or Unity plugins (among others) because the IDEs notify the underlying Plastic plugin about changes to files, renames, moves and the plugin simply performs the controlled operation underneath.

Usually, you will simply use a combination of controlled and not controlled changes; most likely you'll simply modify files, but sometimes you'll do deletes or moves from the Plastic GUI. Plastic is able to deal with both types of changes simultaneously.

A note about move detection and IDEs

The Visual Studio version control integration has a well-known issue notifying moves to the underlying version control when moves happen between different projects.

That's why when using the Plastic plugin for Visual Studio, Plastic needs to use its move detection code under some circumstances.

This happens with other IDEs too and it is because Git is not very good with move detection, so IDEs tend to forget that other version controls can do it better. Well, we simply do the homework and fallback to our detection system when the IDEs don't do their job.

Advanced change tracking

We are very obsessed with the idea of doing precise change tracking in a transparent form. I mean, even if you don't notify Plastic of anything, our goal is to create algorithms good enough to detect what you did after the fact. We love this idea of version control being transparent so you can simply focus on getting your refactors and changes done with minimum interference from Plastic.

That's why we developed a set of techniques that I think is worth sharing.

Fast change detection with watchers

Walking the full workspace tree to look for changes means reading the entries of each of the directories and comparing size, name and last write times with the workspace metadata ([plastic.wktree](#)).

Doing this walk in a naïve way proved not to be fast enough so:

- In Windows, we take advantage of the [FindFirstFile/FindNextFile](#) APIs to minimize the number of reads.
- In Linux/macOS, we use the same technique the find command uses to traverse the disk fast: we use the [FTS functions](#) [<http://man7.org/linux/man-pages/man3/fts.3.html>].
- We take full advantage of Windows [ReadDirectoryChangesW](#) APIs (a.k.a. File System Watchers) to detect changes in the workspace and reduce directory traversals. This allows the GUI to find changes much faster after the first time, because it focuses only on what changed after the first run.
- In Linux, we use [inotify](#) to speed up change detection. The problem is that inotify is much trickier to configure than its Windows counterpart and normally the user needs to do some tuning to use it. Learn how to [enable INotify in Linux](#) [<https://www.plasticscm.com/download/releasenotes/5.4.16.721>].
- As I write this, we plan to take advantage of [FSEvents](#) on macOS, although they are not so flexible as Windows watchers and impose more limitations.

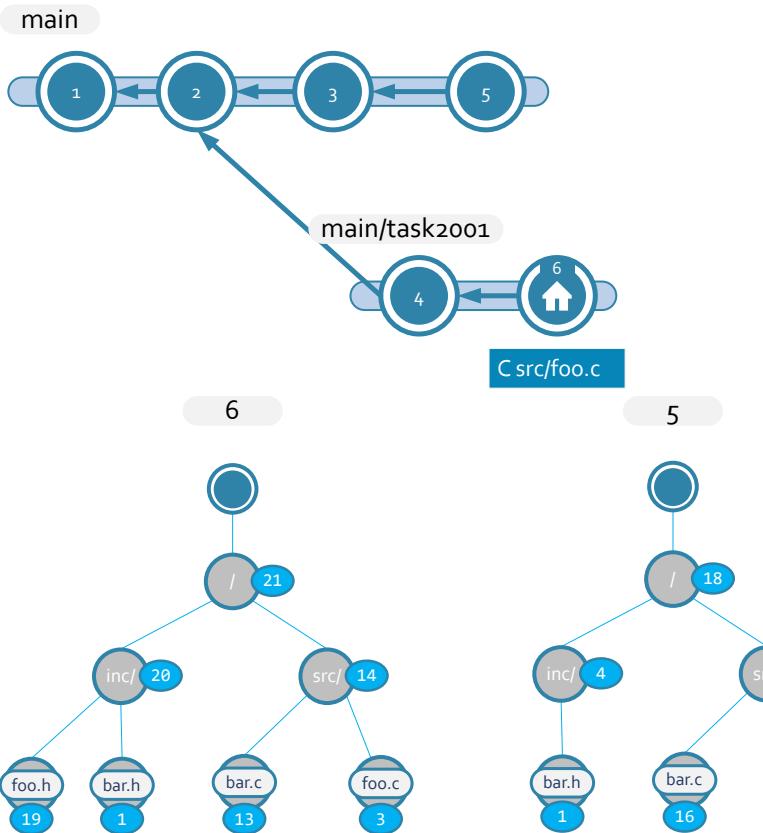
Advanced move detection

In Windows, we can use the USN Journal on NTFS volumes to do precise move tracking. Linux and macOS don't provide an equivalent feature, and that's one of the reasons why we were reluctant to enable it in Windows. This means there is a way to precisely detect moves and renames, detected by the file system itself. The only downside is that many editors don't really rename, but create a new file with the new name and then delete the old.

Switch branch with changes – why it is risky

By default, you can't switch to a different branch (or changeset, or label, I mean, change configuration) if you have pending changes (controlled or not controlled). There is a configuration setting in preferences to allow this switch but it should only be used if you really understand what the behavior will be. And this is exactly what I'm going to explain in this section.

Let's go back to our example repo and suppose we are now on changeset 6 and we modified [src/foo.c](#):



And now, we'd like to switch from changeset 6 to 5 (switch from `task2001` to `main`). Initially, Plastic will tell you that you can't because you have pending changes. If you force the setting, Plastic will let you do the switch and your workspace tree will be as follows:

plastic.wktree	
Name	RevId
<code>/</code>	18
<code>inc/</code>	04
<code>inc/bar.h</code>	1
<code>src/</code>	17
<code>src/bar.c</code>	16
<code>src/foo.c</code>	03

Please note how the revid of `src/foo.c` says that your changes come from revid 03, the one loaded in changeset 6, and not revid 10, the one that you should have after you switched to changeset 5.

Now, suppose you try to checkin `src/foo.c`. Plastic won't let you do the checkin complaining that the metadata of your file `src/foo.c` doesn't match the one loaded by changeset 5.

The actual message is: "The parent revision of the item `/src/foo.c` is inconsistent with the loaded one in

the changeset `cs:5`.

Why does Plastic behave this way?

Well, suppose Plastic lets you complete the checkin: you would be entering changes based on `foo.c` **revid 03**, and completely overriding changes done in revid **10**. You would be losing changes! It would be like copy/pasting a file, overwriting your local changes, and doing checking.

We think that Plastic must prevent data loss at all costs, and that's why it doesn't allow you to work this way.

It might be a little bit confusing if you encounter this situation, but the reason is to avoid losing changes at all costs.

What is the point of allowing you to optionally switch workspace configuration while having pending changes? Well, we heavily considered the option to simply disable such behavior (as Git does), but we are aware of cases where it can be useful:

- You have a config file in version control and you modify it to do some tests.
- Then, you want to test this config on a different branch.
- Well, it is very useful to simply switch branches and test. But, the point here is that you'll never checkin these changes, at least not outside the branch where they were originally made.

What if you really want to override changes?

If you really want to modify changeset **5** with changes coming from **6** without using merges, you can shelve `src/foo.c` and apply those changes to the new configuration and checkin.

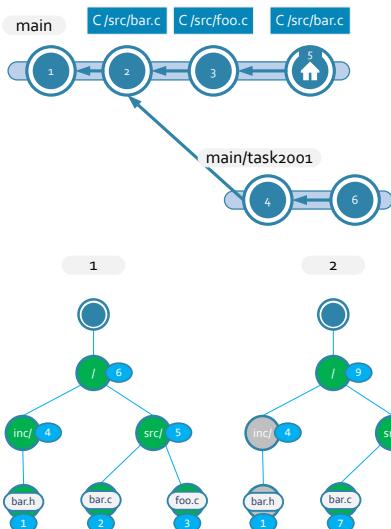
Well, you might say, but this will "merge the changes", not simply override them. Yes, that's true. If you really want to override `src/foo` in changeset **5** with your changes made while working on a different branch, then you have to manually copy the file, switch, then overwrite. I know this is not easy, but what you are doing is really not a recommended practice, it should be an exception case that you shouldn't perform often, and that's why we think we shouldn't make it easier 😊

Full workspaces are always ready to merge

Full workspaces are always in sync with a given changeset. I mean, they can have local changes based on that changeset, but will always be in full sync with it.

What does this mean?

I'm adding again the branches and trees we are using all through this chapter:



What I mean by full sync is that if your workspace is in changeset 5, its metadata has to resemble what was loaded there. You must be loading the correct `bar.c`, `foo.c` and `bar.h`. A configuration like the following will be valid:

plastic.wktree	
Name	RevId
/	18
inc/	04
inc/bar.h	1
src/	17
src/bar.c	16
src/foo.c	10

And, of course, `bar.c`, `foo.c` and `bar.h` could be heavily modified but always based on the right revisions loaded from changeset 5.

If you were in sync to changeset 3, then your workspace tree would exactly load the revisions on that tree, and your workspace changes would be done on top of those versions.

What I mean is that a configuration like the following is not valid in a full workspace:

plastic.wktree	
Name	Revid
/	
inc/	
inc/bar.h	01
src/	
src/bar.c	16
src/foo.c	03

And, it is not valid because you can't be in sync with a changeset while loading revid 16 of `bar.c` and revid 03 of `foo.c` because such a configuration never existed in the repo. See what I mean? Being in full sync means that what you have on disk (before changes, of course) resembles exactly a changeset on the repo.

Well, you may say, you can have `cloaked.conf` in place and then avoid the update of certain files. And yes, that would be correct, but those files would be always updated to resemble the workspace configuration during the merge if needed.

And I already mentioned merge, which is the key for this way of working together with what we saw in "Switch branch with changes": full workspaces are always ready to checkin to the head of its loaded branch, and always ready to merge, that's why:

- They must be in full sync with a given changeset, which means its metadata matches the one in the changeset.
- There can't be files loaded from a different changeset, because otherwise they won't be checkedin (as we saw in the previous section).
- And it won't be possible to perform merges if they workspace is not in full sync either.

And now is when we put together what we learned in the merge chapter with what we just learned about workspace metadata; suppose you want to merge from `task2001` to your workspace based on `main` but with the configuration marked as wrong before: the one loading `bar.c` revid 16 and `foo.c` revid 03.

How can you calculate the common ancestor? You can't.

Common ancestors are calculated based on per-changeset merge tracking, and since your current configuration doesn't match any changeset, there is no way to find where the "destination changeset" is, and hence no way to calculate the common ancestor. If we assume your workspace is on changeset 5 despite these changes in `bar.c` and `foo.c`, the resulting merge would be wrong because it would incorrectly merge those files due to wrong common ancestors, potentially losing changes.

How can you actually lose changes? Well, you are loading `foo.c` revid 03, which was created on changeset 1, but merge tracking will assume you have revid 10, created on changeset 3. You will basically delete all changes made after changeset 3 on `main` if Plastic allowed you to merge from any branch with changes in `foo.c` while having the mentioned inconsistent configuration.

This is the reason why Plastic is always so strict about keeping full workspaces in sync. That's why you need to update before checkin (even if it happens automatically as described in "Conflicts during checkin - working on a single branch") to always keep your workspace in sync to a given changeset and always stay ready to merge.

There is a way to break this synchronization and create more flexible configurations: this is what we call

partial workspaces and we'll see them in depth in the next sections.

Partial workspaces

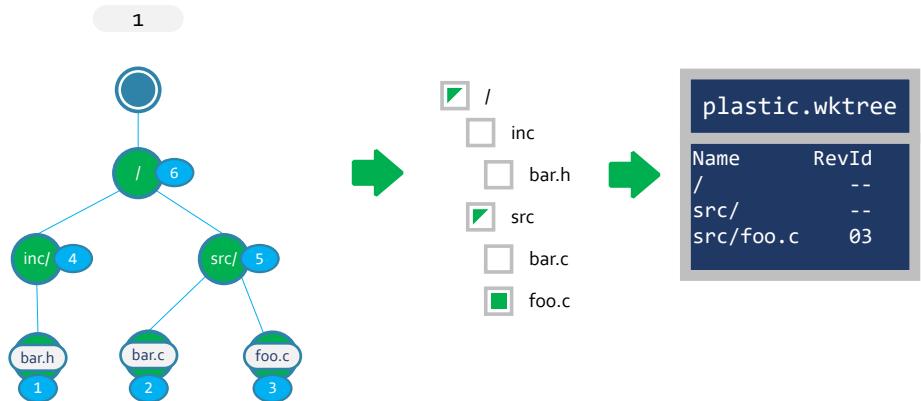
We created partial workspaces for Gluon, the GUI for artists in game development and indeed anyone not dealing with code and mergeable files. Gluon and partial workspaces (and the `cm partial` command) are great to deal with documentation (I'm using Gluon to checkin every single change to this book while I write), images, diagrams, design documents, excels, reports, and any initially unmergeable files that you modify on a single branch.

Gluon and partial workspaces are typically used in combination with file locking and checkouts to prevent concurrent changes on unmergeable files. It is possible to use Gluon with text files that can indeed be merged, but I'd say it is not the main scenario for Gluon and partial workspaces.

Configuring your partial workspace

After you create your partial workspace (typically using Gluon, although you can also simply create a regular workspace from CLI and then run `cm partial configure`), the next step is to configure it. Configuration is basically telling Plastic what files you want to download and track, and it means the workspace will behave totally different than a full one.

I'm going to start with the example we were following during the chapter, and show the trees on the repo and how everything is mapped to the workspace.

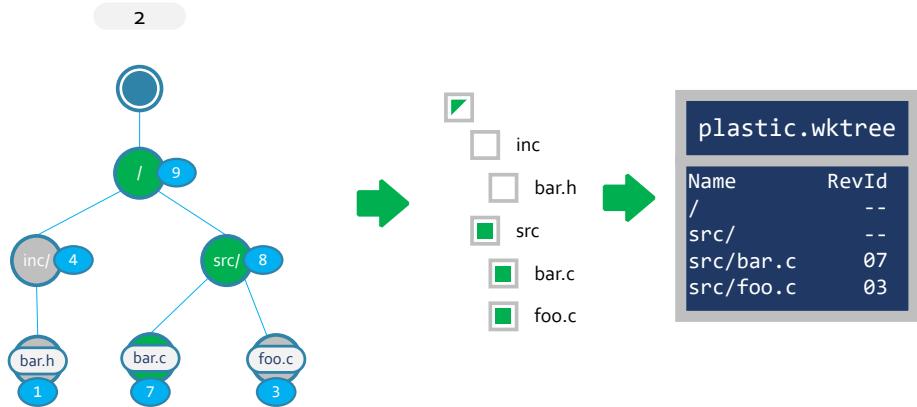


The figure shows how we start working on our partial workspace when the only changeset that existed on the repo was changeset 1. Then, we configure the partial workspace to download only `src/foo.c`, and the workspace metadata will reflect that; it just loads `foo.c revid 03`. Please note how I deliberately omit revision numbers for directories.

Also note how the `src` directory is "partially checked" during configuration. This means not all its children are loaded, and it will have an impact on how the update operation works in partial workspaces. More on that later.

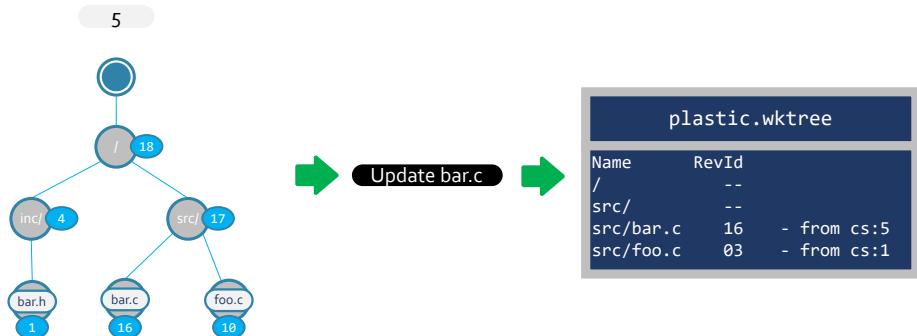
Partial workspaces aren't in sync with a given changeset

Then, after a while, the repo evolves to changeset 2 (someone else did a checkin) and then we decide to configure `bar.c` as follows:



Note how the `src` directory is now fully checked because all its children are fully checked. Our local metadata now loads `bar.c` revid 07 and `foo.c` revid 03, so we could consider our workspace is synced to changeset 2... although that won't last.

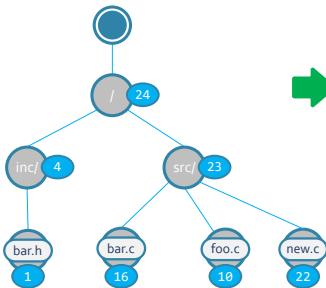
The repo continued evolving and it is now at cset 5. And then, we decide to update just one of the files we have: we only update `bar.c`, but we don't download the new changes of `foo.c`.



At this point we couldn't draw a "house icon" on any changeset on the Branch Explorer because the workspace is not "at any given changeset" anymore. What we have in the workspace is a mix of things that are on the repo, but they are not together in the same way they are in the workspace at any changeset.

Then, the repo continues evolving and it reaches changeset 7. At this point, we run a full update and the new file `src/new.c` will be downloaded, together with the new version of `foo.c` that we refused to download before. The file `new.c` is only downloaded during the workspace update because the `src` directory is "fully checked". If it wasn't fully checked, new files wouldn't be downloaded during the update. At this point, we could say our workspace is again in sync with changeset 7... although this is not a common scenario with partial workspaces, which tend to stay "not synced to a given cset" for a long time.

7

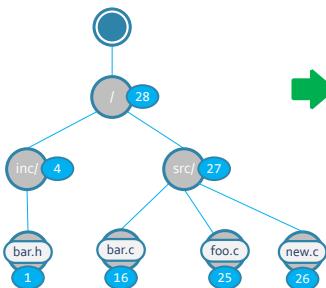


Wk update will download src/new.c because src is fully checked

plastic.wktree	
Name	RevId
/	--
src/	--
src/bar.c	16 - from cs:5
src/foo.c	10 - from cs:7
src/new.c	22 - from cs:7

Changeset 8 brings new changes for `foo.c` and `new.c`, but we only decide to update `new.c`. Our workspace isn't in sync with a changeset anymore.

8



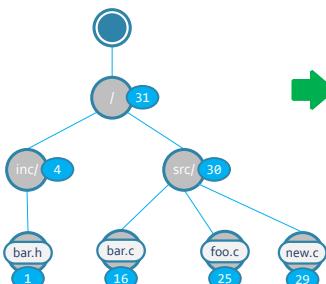
update new.c only

plastic.wktree	
Name	RevId
/	--
src/	--
src/bar.c	16 - from cs:5
src/foo.c	10 - from cs:7
src/new.c	26 - from cs:8

Partial workspaces can checkin without downloading new changes

Then, changeset 9 introduces even more modifications (made again by someone else working on the repo), but at this point we don't do any updates. But, we modified `bar.c` locally.

9



No update in the wk.
bar.c is locally modified

plastic.wktree	
Name	RevId
/	--
src/	--
src/bar.c	16 - from cs:5
src/foo.c	10 - from cs:7
src/new.c	26 - from cs:8

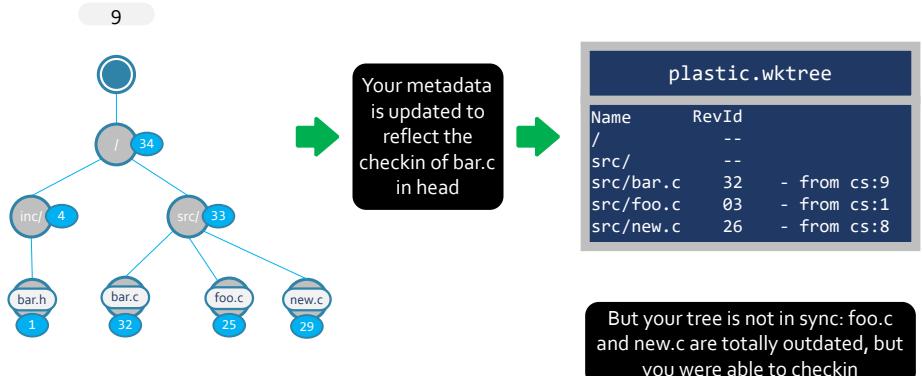
Now, with our workspace clearly not in sync with any changeset, we'll try to checkin `bar.c`. The partial workspace can do that without requiring a previous update to be "in sync with head".

The following figure shows what happens after you checkin `src/bar.c` from the partial workspace. A new changeset is created in the repo (changeset 9 this time) but the local metadata is only updated for `bar.c`,

while the rest stays how it was.

We basically just created a new changeset on the branch, while our workspace doesn't have the same file versions than the new head changeset we just created. This flexibility is the key benefit of partial workspaces.

Checkin changes to bar.c



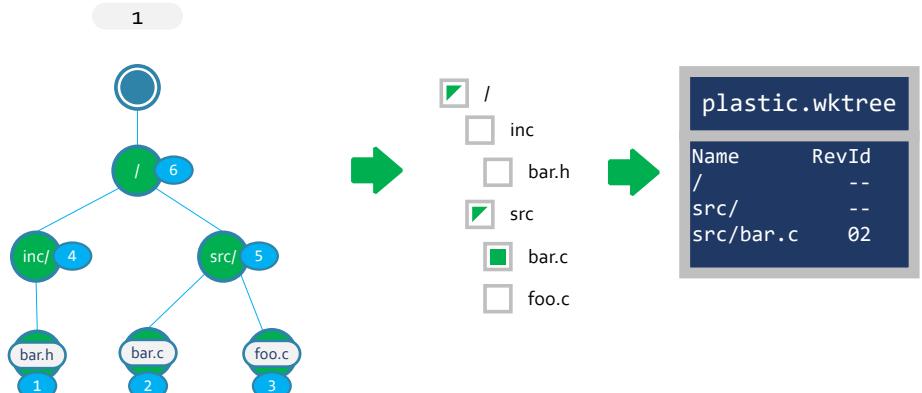
Of course, don't try to perform branch merges on partial workspaces. It wouldn't be possible to merge to the current workspace status because it doesn't resemble any changeset configuration and hence merge tracking wouldn't work.

Other than that restriction, partial workspaces are very useful for per-file version control workflows.

Fully checked and partially checked directories

During the last few sections, I introduced the meaning of partially checked compared to fully checked directories in partial workspaces configurations, but I'm going to explain them here in detail.

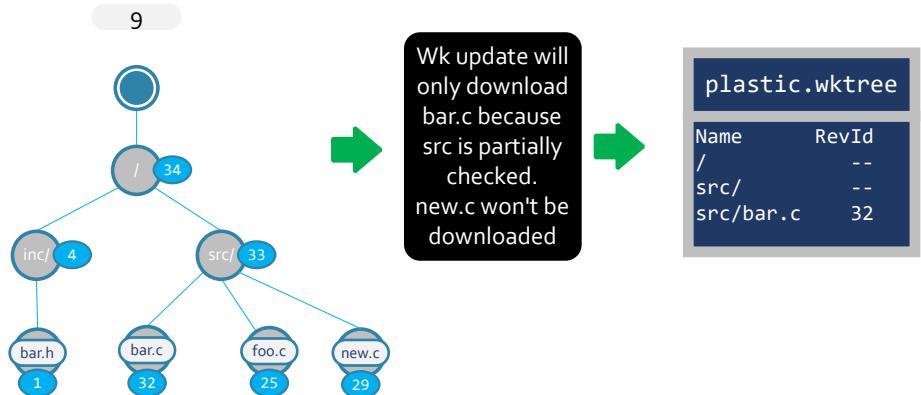
Consider we configure our workspace in the following way:



Note that since I didn't select all the children of 'src/', it is partially checked.

Then, after a while, the repo evolved and new changes to `bar.c`, `foo.c` appear, and also a new file under

src. Will the new changes be downloaded if I simply update the workspace?



The answer is no. Only the configured file **bar.c** will be updated to the latest version. Since the **src** directory is partially configured, new files under it won't be downloaded.

Partially configured directories are designed this way to have a way to avoid new contents from being unexpectedly downloaded. If you want update to download "anything new under a given path", ensure this path is "fully checked".

How to convert a partial workspace into a full workspace and vice versa

Suppose you are working on a partial workspace and for whatever reason (like doing a merge) you need to convert it into a full workspace. How can you achieve that?

Easy.

Simply run a full `cm update` or open the classic Plastic GUI and run an update.

The update will synchronize the workspace with the head changeset of the branch you are working on.



This operation can potentially download lots and lots of GBs depending on the size of your repo.

Now, how to convert a full workspace to partial? Very easy too; simply run `cm partial` configure or open Gluon and run a configure and the workspace will be converted.

Files that require merge during checkin

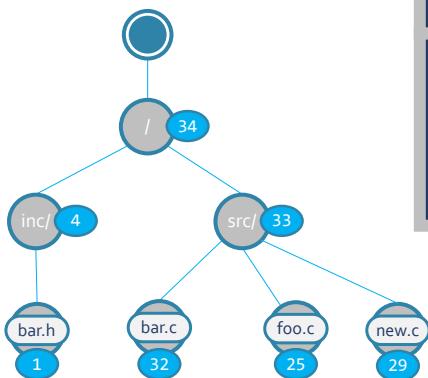
I said that partial workspaces can't be used to do merges, but I referred to merges between different branches, since we added support to deal with files that need merge during checkin.

Consider the following situation:

main



9



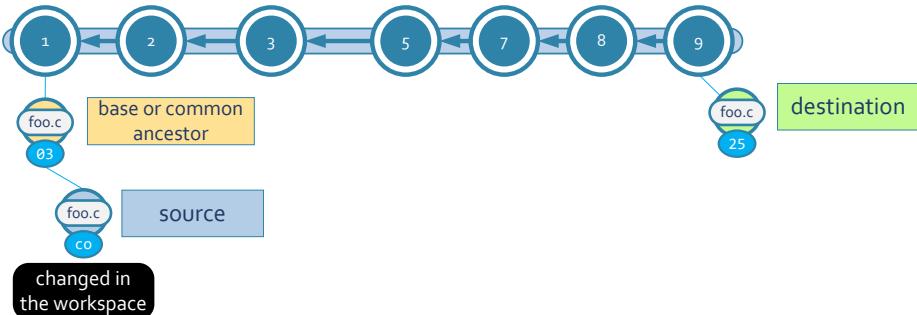
plastic.wktree	
Name	RevId
/	--
src/	--
src/bar.c	32 - from cs:9
src/foo.c	03 - from cs:1
src/new.c	26 - from cs:8

- We see how the repo evolved and now we have up to changeset 9.
- We see the tree for changeset 9.
- And we see the current configuration of the partial workspace: only `bar.c` is "in sync" with changeset 9, but `foo.c` and `new.c` are not.
- I didn't draw a "home icon" on any changeset in the Branch Explorer diagram because the workspace is "not" in a given changeset, as you know by now. The workspace has parts and pieces from different changesets but it is not clearly in sync.
- Finally, I marked `src/foo.c` in a different color to reflect it was locally modified.

Now, we want to checkin `src/foo.c`. What is going to happen?

Plastic is going to merge your local changes of `foo.c` (made based on revid 03) with the latest `foo.c` revid 25 on head (changeset 9). Actually, in single branch merges, it is possible to simulate per-file merge tracking and that's what Plastic does here.

Let me explain in more detail with a diagram:



Plastic accurately assumes that:

- The file changed in the workspace is the source contributor of the merge.
- The file in head is the destination contributor of the merge.
- The parent of the locally modified revision (the one it started from) is always the common ancestor. Since plastic.wktree knows that the original revision for `foo.c` was revid `03`, we have the common ancestor.

Now 3-way merge can operate and resolve the possible conflicts.

Of course, a merge link can't be created because this merge happens at a "file level" and as we discussed in great detail in the merge chapter, Plastic track merges at a changeset level. This is not a big deal anyways, since users working with these types of merges don't want to see merge links anyway.



By the way, this scenario wouldn't have been possible if the user locked `foo.c`, because then no new revisions would have been created and the merge would be avoided. Or, if once changeset exists, the user tries to lock his old copy of `foo.c`. Plastic will never allow him to lock an old version. In this case, of course, a `.c` file is fully mergeable, so locking is not the way to go and that's why we added merge support for these merge needed on checkin situations.

Didn't you say per-file merge tracking wasn't possible?

Yes, in "[Changeset-based merge tracking](#)" we covered in great detail how merge tracking works, and how merge links are only set between changesets, and hence it is not possible to merge just some files from a branch, skipping the others, and then expect to merge those skipped files later.

But partial workspaces use a small "hack" to merge individual files during checkin. The algorithm calculates the common ancestor without using any merge tracking. Due to the linear nature of the scenario (single branch), we can assume that the common ancestor is always the parent of the file being checked in.

This assumption can't be made in any other circumstances when multiple branches are involved (otherwise, the entire merge tracking calculation wouldn't be needed at all 😊).

Xlinks in partial workspaces

There is one thing to keep in mind while using writable Xlinks in partial workspaces and it is that they are always updated to latest on the destination branch of the Xlink and the checkins always placed in head.

Let me explain what this means.

Remember that writable Xlinks are like special symlinks to the root of a given repo in a given changeset. If your Xlink points to a repo in cset `10`, the subdirectory is loaded with the content of that repo in changeset `10`.

But when using partial workspaces, the workspace will always try to download whatever is latest in the branch that contains changeset `10`.

This is because partial workspaces are designed for a very simplistic single branch operation where every time you update you get whatever is latest on that branch, so we applied the same principle to Xlinks.

In full workspaces, the workspace is in sync with a given changeset, so it is very easy to know where the

Xlink points to. In full workspaces, the metadata of every directory is tracked, and we know which is the revid it points to.

But, in partial workspaces Plastic ignores the revid of the directories, because most likely, as we saw in the previous sections, you won't be loading something consistent with any revid of a directory, since you can have outdated files together with up-to-date ones in the same directory. For this reason, it wouldn't be easy to know which version of the Xlink we should be loading at each moment. And hence, we simply decided to make Xlinks point to the head of the branch in partial workspaces.

HOW TO LEARN MORE

By this point, you have gained a great understanding of general version control patterns, practices and internal workings. And you also learned a lot about the Plastic SCM specific way of working.

My goal here is to point you to some good places to learn more.

More about Plastic SCM

There are two key places to learn more about Plastic:

- The **documentation site** [<https://www.plasticscm.com/documentation>].
- The **blog** [<http://blog.plasticscm.com>].

Materials to learn more essential info

Here, I'm going to highlight the main guides in our documentation that you should know to obtain a deeper knowledge of Plastic:

- **Administrator's guide** [<https://www.plasticscm.com/download/help/adminguide>]. This guide covers everything about configuring the server, installing, setup locks if needed, doing backups, archiving revisions, configuring files and licenses. This is a reference more than a book to read cover-to-cover.
- **Security guide** [<https://www.plasticscm.com/download/help/securityguide>]. If you configure an on-premises server, then ensure you take a look at this guide to learn how to perform basic security actions. This is a short guide, full of examples, that you can use to solve specific cases.
- **Gluon - Version control for non-developers** [<https://www.plasticscm.com/download/help/gluonguide>]. If you need to work on a single branch with locks, ensure you take a look at this guide.
- **Triggers** [<https://www.plasticscm.com/download/help/triggersguide>]. Use this guide if you need to customize the behavior of Plastic. You can enforce policies like ensuring branches follow a given naming convention, and many others.
- **Cloud** [<https://www.plasticscm.com/download/help/cloudguide>]. Learn how to use Plastic Cloud Edition or add a Cloud extension to your Enterprise Edition. It explains how our Cloud service works.
- **Find** [<https://www.plasticscm.com/download/help/cmfind>]. Plastic can query branches, changesets, and other types of objects. This guide explains how to run these queries.
- **Command line** [<https://www.plasticscm.com/download/help/commandline>]. Learn how to use the command line to perform some common operations.
- **Xlinks** [<https://www.plasticscm.com/download/help/xlinks>]. Learn how to do component-oriented development with Xlinks. This guide extends what we covered in this current book in more detail.

Git interop

There are two reference guides to learn how Plastic interoperates with Git:

- **GitServer** [<https://www.plasticscm.com/download/help/gitserverguide>]. This guide explains how to setup a Plastic SCM server in such a way that Git clients think it is a Git server. It is very good to help transition other team members from Git or to interoperate with third-party systems that only speak the Git protocol.
- **GitSync** [<https://www.plasticscm.com/download/help/gitsyncguide>]. Learn how to use Plastic as a Git client and push/pull to GitHub or any other Git service.

References

Use the following only as reference; they are not meant to be read like a book but just describe how certain features work. Armed with the knowledge in this current book, most of what the guides cover will be straightforward to master.

- **Graphical user interface** [<https://www.plasticscm.com/download/help/guiguide>]. A reference of the features in the GUIs.
- **Task and issue trackers** [<https://www.plasticscm.com/download/help/taskandissuetrackers>]. How to configure Plastic to interact with Jira, Polarion and a few others. Includes info on how to create your own integrations with issue tracking systems.
- **External parsers** [<https://www.plasticscm.com/download/help/externalparsers>]. How to develop support for new languages for the Plastic semantic features.

DevOps with Plastic

As we covered in the "One task - one branch" chapter, the entire task-oriented workflow is oriented to implement DevOps. But besides using task branches, you need a way to get the branches merged and deployed.

There are two main ways to achieve this with Plastic: using mergebots (where Plastic drives the CI and decides when to merge and build) or delegating the leadership to a Continuous Integration system.

DevOps driven by mergebots

- **mergebot: the story of our DevOps initiative** [<https://www.plasticscm.com/download/help/devopsinitiative>]. Explains what mergebots are and how they can help you implement DevOps.
- **Add a mergebot to your repo!** [<https://www.plasticscm.com/download/help/hireamegebot>] A practical example and tour through a working DevOps implementation driven by a mergebot.
- **Configure mergebots using config files** [<https://www.plasticscm.com/download/help/configuremergebotsconfigfiles>]. Explains how to manually configure mergebots without WebAdmin assistance.
- **Plastic SCM DevOps: Custom plugs** [<https://www.plasticscm.com/download/help/devopscustomplugs>]. Explains how to develop plugs: the connectors between mergebots and system such as Slack, email, etc.
- **Plastic SCM DevOps: Custom mergebots** [<https://www.plasticscm.com/download/help/devopscustommergebots>]. Explains how to develop your own bots.

DevOps driven by the CI system

- **A DevOps Primer** [<https://www.plasticscm.com/download/help/devopsprimer>]. An explanation of what DevOps is, how we understand it, and how we implement it with Plastic.
- **DevOps with Bamboo and Plastic** [<https://www.plasticscm.com/download/help/devopsbamboo>]. A practical example explaining how to implement a DevOps cycle with Plastic and Bamboo.
- **DevOps with Bamboo: connecting to Jira** [<https://www.plasticscm.com/download/help/devopsbamboojira>]. Extends the previous example and explains how to take advantage of the extra info provided by the issue tracker.
- **DevOps with TeamCity** [<https://www.plasticscm.com/download/help/devopsteamcity>]. An alternative implementation of DevOps, this time using JetBrains' TeamCity CI system.
- **CyberFlex - Jira, TeamCity and Plastic integration** [<https://www.plasticscm.com/download/help/cyberflexjirateamcityplastic>]. One of our customers shares their CI implementation.

Blog highlights

We regularly publish content on our [blog](http://blog.plasticscm.com) [<http://blog.plasticscm.com>] and I'd like to highlight a few blogposts that are worth reading:

- **Plastic vs. Git** [<https://www.plasticscm.com/download/help/plasticvsgit>] shows you how Plastic compares to the now ubiquitous version control and what we try to do better.
- **All the software we write** [<https://www.plasticscm.com/download/help/allsoftwareewrite>]. If you want to learn more about the work we do and the software we develop.
- **The story of Jet: Plastic's super-fast repo storage** [<https://www.plasticscm.com/download/help/jetstory>]. It explains why we now use Jet as the default storage and why it is faster than the other alternatives we had.
- **Diff math** [<https://www.plasticscm.com/download/help/diffmath>]. It explains some uncommon facts of very common diffs.
- **Track refactored code across files with Plastic SCM** [<https://www.plasticscm.com/download/help/trackrefactoracrossfiles>]. It explains some of the semantic features in Plastic SCM.
- **Using history to better explain branch differences** [<https://www.plasticscm.com/download/help/historytoexplainbranchdiff>]. Delves into how Plastic enriches diffs with info from the file history.

How we work

In 2017, we wrote a series of blogposts explaining how we develop our products, including the patterns and practices. The series was written before our move to mergebots but most of what it tells is still relevant:

- **How we do trunk based development with Plastic SCM** [<https://www.plasticscm.com/download/help/trunk>]
- **To deploy versus to release** [<https://www.plasticscm.com/download/help/todeployvstorelease>]
- **Trunk-based development blends well with task branches** [<https://www.plasticscm.com/download/help/trunkandtaskbranches>]
- **How we do trunk-based development: answering frequent questions** [<https://www.plasticscm.com/download/help/trunkfaq>]
- **Plastic SCM development cycle - key practices described** [<https://www.plasticscm.com/download/help/>]

Great books to read

I already mentioned some of them in previous chapters, but I think it is worth listing them together here. For younger readers, I wonder if you would prefer a compilation of blogs instead, but I think there's no better way to learn than taking advantage of someone who already collected good knowledge and put it together, instead of having to do the exercise by yourself. Not that reading online texts are bad, of course, but some of the greatest foundations are easily obtained from a few good books.

- **Software Configuration Management Patterns: Effective Teamwork, Practical Integration** [<http://www.scmpatterns.com>] by Steve Berczuk with Brad Appleton. It is the bible of patterns and practices, and although it was written in the early 2000. It is still unsurpassed by any other books in the matter.
- **The DevOPS Handbook. How to Create World-Class Agility, Reliability, and Security in Technology Organizations** by Jez Humble, Gene Kim, John Willis, Patrick Debois. It is an easy-to-read book (unlike the previous title on Continuous Delivery by Humble), entertaining and full of game changing experiences. If you need to understand why automating the last mile matters, or why DevOps in general makes sense from a business and technical perspective, this is your book.
- **Clean Code: A Handbook of Agile Software Craftsmanship** by Robert C. Martin. It is the team's favorite text on how to write code. I must admit it is not my personal favorite but it is certainly a good one and since the team here at Plastic strongly recommends it, I had to add it to the list ☺
- **Implementation Patterns** by Kent Beck. This one is the book I prefer over Clean Code. I consider it more serious and solid. You might wonder why I recommend books that are not strictly related to version control. Well, the reason is that I view version control as a part of a wider discipline. I like to see version control, and of course Plastic, as a tool not only to control projects, but to produce better, more maintainable, easier to read code. And well, to achieve that goal it is good to understand what some experts consider good code ☺
- **Agile Project Management with Kanban** by Eric Brechner. I loved this book and it was the one that triggered us to jump from Scrum to Kanban. I read a bunch of titles on Kanban but this was the one I really considered great. Of course, this might be a very personal choice, but I strongly recommend it. Task branches and DevOps blend extraordinarily well with Brechner pragmatic view of Kanban.
- **Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing** by Elisabeth Hendrickson. This book helped us better understand the role of manual testing and served as the basis of our validations. I always preferred automatic tests better than manual, but only because I saw many teams doing manual tests that were just scripted procedures that could be easily automated. Exploratory testing really gets the best of the human tester by making them use their creative side instead of constraining them to follow procedures.
- **Code Complete: A Practical Handbook of Software Construction** by Steve McConnell. This is my all times favorite book on how to write software. It was originally written in the 1990s, but to me it is still one of the greatest books on the subject. It shows its age especially for young readers, but I would still recommend it as the best one. It is not only about code; if you want to clearly understand the difference between a walkthrough and a formal code review, you'll find it in McConnell's greatest text.
- **Agile Project Management with Scrum** by Ken Schwaber. One of the originals textbooks on the matter. I asked all my students in the university to read chapters 1 and 2 as the best and shortest descriptions about how Scrum works. And, it used to be a mandatory read for all newcomers to the team here too.
- **Rapid Development: Taming Wild Software Schedules** by Steve McConnell. I'm probably showing

my age here, but Mr. McConnell nailed it with this book where he plotted a ton of great practices to develop software faster prior to the agile explosion. It is still a great book to understand how to work in iterations and avoid the classic (and everlasting) pitfalls of project management.

- **Agile Estimating and Planning** by Mike Cohn. If estimating is something you need to do (you never know anymore since many teams, including ourselves, pivoted to a more organic way of working and delivering software), then this book will explain lots of great things. It includes chapters on how to prioritize features based on value, and many other interesting topics I later found repeated again and again on other books.
- **Software Estimation: Demystifying the Black Art** by Steve McConnell. Well, it is obvious by now who my favorite author is. Books on software estimation tend to be... well, ultra-boring. Cohn's book and this one are my two favorites. T-shirt sizing and techniques like PERT to calculate best, worst and most likely estimation and combat our natural optimism as developers are greatly covered here.
- **Manage It! Your Guide to Modern Pragmatic Project Management** by Johanna Rothman. Any book by Rothman is great to read, including the one on not doing estimations ("Predicting the Unpredictable"). I used "Manage It!" as one of the textbooks for the project management classes during my six years as university professor (together with classics like "Mythical Man Month" that I won't list because while no list is good without this book I don't want to add it... well, I did it 😊). It is fully packed with great advice on how to steer projects.
- **Peopleware: productive projects and teams** by Tom DeMarco and Timothy Lister. There are many books on how to make teams shine, but this is the original. I arranged our first office (and even the current one) based on the advice in this book. Much newer writings like "*It doesn't have to be crazy at work*", while good, don't say much Peopleware did decades ago.

The list could go on and on but I just wanted to include the books that I strongly recommend; the ones I believe helped me move forward as a software developer and I recommend to the people that I work with. Probably, "Pragmatic Programmer" should be on the list too.

APPENDIX A: HISTORY OF PLASTIC SCM

It is always comforting to share why we decided to create Plastic and how everything started. The risk is to bore you with a second hand story 😊.

It all started with a dream... and a challenge

It was back in 2002 when a pair of young programmers dreamed about a better version control system while working for Sony in Belgium, making set-top boxes for what would become digital television.

At that time, it was Clearcase that was the dominant player in the large corporations. It was good, powerful, but difficult to use, super hard to configure, slow as hell when not properly setup (and it almost never was) and extremely expensive.

The other options were mainly Perforce, which was already becoming a myth of its own, Subversion, CVS was still around, and a little bit of Visual Source Safe in the Windows world.

We wanted to come up with something as good as Clearcase (what captured our imagination was those dynamic views implemented as versioned virtual filesystems) but more affordable than Perforce and of course easier to use and install than any of the alternatives.

Good branching and good merging had to be there too. And, the oldest of you will remind that it was not an easy proposal at the time, because many were still singing the "branches are evil" song... exactly with the same software years later they used to say "branches rock" (ahem, Perforce).

Getting real

It is one thing to toy with the idea of creating a new product, and a very different one doing what is necessary to make it happen.

For us it happened in 2004, after a couple of years thinking about options, with some peaks of intense activity and valleys of total calm. We decided to write a business plan and try to find some investors. It was well before the *kick-started is cooler* wave we are into now. And, we really thought: if nobody is going to put money into it to build it, it means it is not worth the effort.

So, we learned about business plans, carefully crafted formats (all so out-of-date by these days lean standards), presentations, elevator pitches, numbers and all that. Of course, time proved it was all entire wrong 😊.

A dream comes true

I quit my previous job around March 2005 to focus on a version control consulting business full-time. Long gone were the days of Sony, and this move happened after a couple of years working as CTO for a small company building and selling an ERP.

We already had some contacts with potential investors since late 2004, lots of pitches, tweaks to the plan, and also participated as speakers in a few conferences, trying to build us a name as version control experts.

It was a sunny morning at the end of June 2005 and we had a meeting in the 23rd floor of the tallest building of the city. We had a few meetings before with Mr. VC. (He's quite shy so I'll better not share the name, and just use VC for venture-capitalist.) He reviewed the numbers and he was enthusiastic about the project, but also had a huge amount of concerns. That morning we reviewed a few more numbers with the help of his assistant and he suddenly said: "well, I just want you to know this is going to happen". I was speechless.

Years later I tend to take it for granted, but when I take a moment to remember that morning when we raised our first almost-half-a-million euros to build Plastic with nothing more than a few numbers, documents and not a single line of code... yes, it was a dream come true.

What came after that was a very intense year. The company was officially created in late August. The office setup at the beginning of September. The first two engineers (they are still here) joined in October. Then months of hard work, but with that feeling every morning was Christmas and you had a bunch of presents to open. Good old days.

We started using Plastic internally on February 23rd 2006. Not long after, we had our first public presentation to a local audience. And Plastic 1.0 was officially released in November 2006 at Tech Ed Developers that luckily happened to be in Barcelona that year, only a 6 hour drive from our place.

Códice and Plastic

We named the company "Códice Software" because we wanted to have something that sounded Spanish in the name (that "ó") and at the same time we were fascinated by the beauty of code, and a codex (código) was sort of a way to bring something ancient into a modern day technology. (Well, it was also when the book and movie the Da Vinci Code came out, so... you name it).

With Plastic it was different; we wanted a name that was easy to say in any language, didn't have a bad meaning, and somehow had a connotation that the product was flexible and a container to protect the code. Plastic seemed right. Of course, the .com domain was taken and SCM was a good choice. SCM as in Software Configuration Management, which was the serious thing to do back in the ALM days. Now, I prefer to think of SCM as simply meaning Source Control Manager, but that's a different story.

Fast forward 15 years and a recent awareness campaign of the National Geographic that reads "Planet or Plastic" makes me wonder if it was the right name ☺

Plastic 1.0, 2.0 and first international sales

2007 and 2008 saw our first customers coming in. Well, the first paying one actually happened before the end of 2006, just a few months after the official launch if I remember correctly, but I think he didn't actually pay until a few months later, already in 2007.

As I said, we started getting visitors, evaluators, doing all kinds of wrong things on the sales and marketing side, and closing some deals.

I always thought Plastic was too expensive for small teams back then, and that it would have been better

to give away tons of free versions, but we had bills to pay and cash-flow was king. Yes, probably short-sighted but I'm not sure we had any options.

We got our first customers outside Spain in late 2007 and our first salesperson in the US in early 2008.

We actually achieved Capability Maturity Model Integration (CMMI) Level 2 in 2007 and became the first Spanish SME to do so, combined with the fact that we were one of the few that achieved that certification using SCRUM. We grabbed some local headlines.

VC time

We soon started to realize that we would need additional funding. We were about seven to eight people in 2008. We tried several options and then met, by total chance, Bullnet Capital, a venture capital firm focused on high-tech companies. They liked what we were doing way out in nowhere in Boecillo Park and things started to move forward.

I remember when we got the go in October 2008 and then suddenly world economy started to crash. Every single day there was bad news, so we were horrified at the thought that we would never get the investment. Fortunately, the events of 2008 didn't have much to do with Bullnet and the money was in the bank in early 2009.

Growing up – the road to 4.0

Since then, we have tried many different things, some went fine, some others didn't. We had a full 5-person sales team in the US in 2010. We tried PR, analysts, direct marketing, targeting enterprises, email marketing...

Plastic was growing up from 2.0 to 3.0 and then the long road to 4.0 started.

I didn't mention it yet, but almost the same month our first business angel decided to invest in us, the first Git was released. A version control platform made by Linus Torvalds himself. Sounded like trouble, and in fact it was.

Versions up to and including 3.0 all implemented per-item merge tracking. The thing deserves its own series on its own, but it basically means that you can merge a single file from a branch, checkin, then merge next. It was super-flexible but a pain for speed: you merge 20k files, you need to calculate 20k trees. This is what Perforce still does and the reason why its merging is slow. And exactly the same reason why we have replaced Clearcases all over the world because of their "hours to complete a big merge". The opposite approach is per changeset merge tracking and that's what Git did, and what we decided to do in mid-2010. It was a huge redesign and a huge risk. Time proved that we were right, because it also allowed us to easily sync with Git, and *speak their protocol*.

Soon after releasing the first betas of 4.0, we got our first +1000 seat operation in South Korea. A mobile phone manufacturer decided to replace their Clearcase with Plastic. We grew up thanks to these folks. 1000 developers on the same small set of repositories (less than 10), all working on a single project. Repos contained everything from the Linux source code (Android) to the quake code (don't know why, a game or a test?) to grow larger than 700k elements on a single working copy.

Plastic graduated as a scalable version control capable of dealing with big loads. It wasn't hard to convince the skeptical after that ☺

A mature solution

2012 to present saw Plastic 4.0, 5.0, 5.4, 6.0, 7.0 and new products spawning like SemanticMerge, where we finally released all our ideas about how to do better merges.

The frantic early days are now long gone, although most of the spirit (and team members who made it happen) are still here.

When we look back, we see how crazy it was when we called a team of ten *large* and how nowadays the dev team is not even notified if a customer with +200 users signs up until the next all-together meeting.

Of course, most of the greatest features and improvements happened in this latter period and are still happening every day. We learned a lot about making things simpler, about product design, testing, optimization, performance, merge cases, merge tools and all the things that, together, make Plastic.

We are still the smallest team in the industry creating a full version control stack, evolving it and ensuring it is worth it compared to the huge alternatives like GitHub/Lab and the likes.

But, from the story telling point of view, I think you get a good idea of how Plastic was created and evolved over the last few years.

APPENDIX B: PATTERN FILES

— Chapter by *Miguel González* [<https://www.plasticscm.com/company/team#miguel-gonzalez>]

Pattern files are plain-text files that contain rules, one per line, to match files according to their paths. There are currently two kinds of pattern files that exist in plastic: the **filter pattern files** and the **value matching pattern files**.

Filter pattern files

These pattern files apply to the workspace tree or pending changes to specify how items should be filtered.

These include:

- Ignored files (`ignore.conf`)
- Cloaked files (`cloaked.conf`)
- Hidden changes (`hidden_changes.conf`)
- Writable/read-only files (`writable.conf` and `readonly.conf`)
 - These two files apply to the same action (set the item as read-only)
 - The first one (`writable.conf`) takes precedence over the second one (`readonly.conf`)

Each of these files contain one pattern per line. Plastic SCM will check the path of each item to see if it's affected by the set of rules. You can comment lines by placing the `#` character at the beginning of any line.



Keep in mind that any filtered directory will filter their complete subtree. This means that if `/src/lib` is filtered, `/src/lib/core.h` and `/src/lib/module/main.c` are filtered as well.

Rule types

Absolute path rules

Allows you to match a single file or directory:

```
# The following would match the *complete workspace tree*:  
/  
  
# The following would match:  
#   * /src/main/test  
#   * /src/main/test/com/package/BasicTests.java  
/src/main/test  
  
# The following would only match:  
#   * /src/lib/server/Main.java  
# But it would skip these:  
#   * /Main.java  
#   * /src/lib/client/Main.java  
/src/lib/server/Main.java
```

Rules with wildcards

You can customize the Absolute path rules with wildcards:

- `*` will match any number of consecutive characters except the directory separator `/`
- `**` will match any number of consecutive characters, including the directory separator `/`
- `?` will match a single character, excluding the directory separator `/`

```
# The following would match:  
#   * /src/lib-3/main.c  
#   * /src/samples/number-3/main.c  
/src/**-3/main.c  
  
# The following would match:  
#   * /doc/public/toc.tex  
#   * /doc/public/chapter-1.tex  
/doc/public/*.tex  
  
# The following would match:  
#   * /src/lib/code.c  
#   * /src/lib/coda.c  
/src/lib/cod?.c
```

Catch-all rules

A subset of the rules with wildcards, these rules are equivalent and they will match the complete workspace tree:

- `/`
- `*`
- `/*`
- `*/`
- `**`
- `/**`
- `**/`

Item name rules

Matches files or directories according to their names.

```
# The following would match:  
#   * /src/lib  
#   * /src/lib/client/main.c  
#   * /src/lib/test  
#   * /references/lib  
#   * /references/lib/libgit2.so  
lib  
  
# The following would match:  
#   * /src/lib/references.c  
#   * /references.c  
#   * /doc/main/references.c  
references.c
```

Extension rules

These rules match files according to their file extension.

```
# The following would match:  
#   * /src/main/java/com/samplepackage/Main.java  
#   * /examples/ReferenceList.java  
.java  
  
# The following would match:  
#   * /bin/client/resources/core.en.resx  
#   * /out/bin/server/resources/networking.en.resx  
^.en.resx$
```

Regular expression patterns

When everything else is not enough, you can fall back to regular expression matching. The only requirement for this is to ensure that the pattern starts with the line start symbol for regular expressions (^) and it ends with the line end symbol (\$).

```
# The following would match:  
#   * /src/doc/sample-3528.txt  
#   * /sample-1.txt  
^.*/sample-[0-9]+.txt$  
  
# The following would match:  
#   * /bin/dir-0xC0FEE/main.c  
#   * /publish/bin/dir-0xDADA/_backend.cpp  
^.*/dir-0x[A-F0-9]+_?/[^\^]+$
```

Include / Exclude

All rule formats we previously discussed work in a **positive** way; if the item path matches the rule, the result is a positive match and Plastic SCM will apply the filter to that item. However, there are situations where you want to apply the rule in a **negative** way, that is, **exclude** the item from the filter. This is particularly useful when you wish to filter a directory but keep some of the files it contains out of the filter.

These rules are identical to the regular inclusion rules, but start with the `!` character.

```
!/src/lib  
!/main/bin/compiler.exe  
!*.  
!references  
!resources.*
```

Pattern evaluation hierarchy

Plastic SCM will try to match the path of an item using the patterns in the file in a predefined way. This means that some pattern formats take precedence over others, rather than processing the patterns in the order that they appear in the file.

1. Absolute path rules that match exactly
2. Catch-all rules
3. Name rules applied to the current item
4. Absolute path rules applied to the item directory structure
5. Name rules applied to the item directory structure
6. Extension rules
7. Wildcard and Regular expression rules

Directory structure matching

"*Directory structure*" means, in the list above, that the rules are applied to the full list of directories above the current item.

For instance, if the current item path was

```
/src/main/java/com/codicesoftware/sample/SelectedItem.java
```

any of these two rules would match its directory structure:

```
/src/main/java  
codicesoftware
```

The first one, `/src/main/java` partially matches the directory hierarchy of the current item. The second one matches the name of one of the parent directories of the current item.

Include / Exclude rules order

Exclusion rules (ones that start with `!`) take precedence over include (or regular) rules, so they are matched **first**.

There are two exceptions to this. The first one applies to **absolute path rules that match exactly** the current item. In this case, the **include** rules are applied first.

For example, if we used these rules:

```
!/  
/src/test/testdata.zip
```

The item `/src/test/testdata.zip` is filtered even if we explicitly prevented the whole workspace tree from being filtered.

The second exception applies to **absolute path rules applied to the item directory structure**. In this case, the most precise rule is the one that takes precedence.

For example, if we used these rules:

```
/src  
!/src/client  
/src/client/bin
```

Files like `/src/build.sh` or `/src/client/bin/output.so` would be filtered, but files like `/src/client/socket.c` wouldn't. This is because `/src/client/bin` is a more precise match than `/src/client`, which is in turn more precise than `/src`. In this case, you can also consider precision as the length of the matched substring.

We'll use this tree in the next few examples:

```
.  
| #build.txt  
| Makefile  
+---doc  
| | #build.txt  
| | Makefile  
| | manual.pdf  
| +---pub  
| | chapter1.epub  
| | chapter2.epub  
| \---src  
| | chapter1.tex  
| | chapter2.tex  
| | toc.tex  
\---src  
| | #build.txt  
| | operations.build.sh  
| | README.txt  
+---client  
| | command_line_args.c  
| | main.c  
| \---gui  
| | window.c  
\---server  
| libgit2.so  
| socket.c  
| socket.h
```

Example 1. Directory structure matching

To see the directory structure matching in action, let's say we'd like to filter all but the `/src` directory, but inside of it filter as well the `/src/client/` directory. Finally, we'd like to keep `/src/client/gui`. Here's how:

```
/  
!/src  
/src/client  
!/src/client/gui
```

This is the resulting tree:

```
.  
\---src  
|   #build.txt  
|   operations.build.sh  
|   README.txt  
+---client  
|   \---gui  
|       window.c  
\---server  
    libgit2.so  
    socket.c  
    socket.h
```

As you see, the longest matching rule takes precedence over the rest when only absolute paths are involved.

Example 2. Exclude / include precedence

If we added this rule to the set above:

```
*.h
```

It wouldn't affect `/src/server/socket.h` because it's excluded by `!/src`, and exclude rules take precedence over include rules.

Example 3. Full path match

There's a way to filter files affected by a exclusion absolute path rule. However, that requires exact path matching.

To filter `/src/server/socket.h`, we just add that particular path to the set of rules:

```
/src/server/socket.h
```

That's the only include rule that takes precedence over all of the rest.

Example 4. Name rules

If there's a recurring file name that appears over and over again, you might want to filter it using just the name:

```
Makefile
```

The rule above will filter all `Makefile` files in your workspace.

Example 5. Wildcards

What if we'd like to apply a more complex filter, such as "all files with `oc` in their names, and a single-letter extension"? Easy, use wildcards!

```
**/*oc*.?
```

This would filter out `/src/server/socket.h` and `/src/server/socket.c` but it wouldn't match `/doc/src/toc.tex` because its extension has 3 characters!

That's also the way to match files whose name starts with the `#` character. As we said before, that's how comments are specified... so how would you filter all those `#build.txt` files?

One way to do it is using **extension rules**:

```
*.txt
```

But, that would filter **all** text files in the tree. To make it more precise, use this instead:

```
**/#build.txt
```

Value matching pattern files

This kind of pattern files assign a value to items that match the specified patterns. It's suitable for functionality that doesn't work in a yes/no fashion, such as assigning the type of a file (binary compared to text).

These include:

- Compression type (`compression.conf`)
- EOL sequence conversion (`eolconversion.conf`)
- File type (`filetypes.conf`)

These files also contain one pattern per line. Plastic SCM will check the path of each item to find out whether it matches any of the rules to return the assigned value.

Lines in these files will contain rule/value pairs. The separator sequence changes across files:

- `filetypes.conf` uses `:` as the rule/value separator

- `compression.conf` and `eolconversion.conf` use whitespace as the rule/value separator

Pattern types are limited to four: file extension, file name, file path and wildcard rules.

You can comment lines by placing the `#` character at the beginning of any line.

Rule Types

Extension rules

These rules are the exact extension of the files to match. This applies to the characters after the last `.` in the file name.

Example 6. compression.conf

```
.png none  
.c zip  
.cpp zip
```

Keep in mind that values such as `.en.resx` or `.in.html` won't match any file because the matching engine will receive `.resx` and `.html` as the item extension for those examples.

In this example, files such as `/theme/images/background.png` or `/wwwroot/img/top1.png` will never use compression, whereas files such as `/src/client/main.cpp` or `/src/lib/core/threadpool.cpp` will always use GZip compression.

Path rules

These rules compare the full path of the current item with the value of the rule. If both match, the process will assign the related value of the current rule to that item.

Example 7. eolconversion.conf

```
/src/main.c auto  
/lib/core/Calculator.cs CRLF  
/src/main/java/com/codicesoftware/sample/Main.java LF
```

These type of rules is very easy to see in action, since the rule matches exactly the item path. `/src/main.c` will use automatic EOL conversion, `/lib/core/Calculator.cs` will convert all its EOL sequences to `CR+LF` and `/src/main/java/com/codicesoftware/sample/Main.java` will convert them to `LF`.

Name rules

Similar to the path rules, the name rules use exact matching of the item name.

Example 8. filetypes.conf

```
header.png:bin  
README.md:text
```

Here, files such as `/my-app/wwwroot/img/header.png` and `/images/header/header.png` would be always set as *binary*, whereas files such as `/README.md` or `/src/doc/README.md` would be always set as *text*. Any other file would have its file type detected from its name and contents.

Wildcard rules

Finally, path rules can be enhanced with the special sequences defined in "Rules with wildcards" in the [Filter pattern files](#) section:

- `*` will match any number of consecutive characters except the directory separator `/`
- `**` will match any number of consecutive characters, including the directory separator `/`
- `?` will match a single character, excluding the directory separator `/`

Example 9. compression.conf

```
/**/images/**/*.* none  
/src/client/core/lib_01.? zip
```

If you apply this compression filter, files like these will never use compression:

- `/wwwroot/dist/images/mandatory-intermediate-directory/img.png`
- `/doc/images/builder/readme.txt`

These other files will always use GZip compression:

- `/src/client/core/lib_01.c`
- `/src/client/core/lib_01.h`
- `/src/client/core/lib_01.o`

Pattern evaluation hierarchy

Plastic SCM will try to match the path of an item using the patterns in the file in a predefined way. This means that some pattern formats take precedence over others, rather than processing the patterns in the order they appear in the file.

1. Path rules
2. Name rules
3. Extension rules
4. Wildcard rules

Example 10. filetypes.conf

```
compile.exe:txt  
/src/main/bootstrap/compile.exe:bin
```

In this example, `/src/main/bootstrap/compile.exe` would be *binary*, but any other `compile.exe` file in the tree (for example, `/build/release/compile.exe`) would be *text*.

Example 11. eolconversion.conf

```
/src/java/**/*.* LF  
.java:auto
```

In this other example, any file under `/src/java/<subdir>/` would have its EOL sequences forcefully converted to LF. For example, `/src/java/main/com/sample/README.txt` or `/src/java/test/resources/res.xml`

Also, all `.java` files in the tree would have its EOL sequences handled automatically, for example `/main/Program.java` or `/src/module/first/Connection.java`

And finally, since exception rules take precedence over wildcard rules, `.java` files **inside** `/src/java/<subdir>` will also have their EOL sequences handled automatically. For example, `/src/java/main/com/codicefactory/writers/PlasticStreamWriter.java` or `/src/java/test/complex/ExtremelyRareScenarioTest.java`.

APPENDIX C: ALTERNATIVE BRANCHING STRATEGIES AND SOLUTIONS TO FREQUENT PROBLEMS

We always recommend task branches blended with trunk development as a very simple and effective way of working, but there are other alternatives worth considering.

Not all teams can directly jump to a practice that requires the presence of effective automated testing and a CI system. Sometimes, tests are not automated, or builds are too slow.

This chapter describes alternative working patterns and how to use them as an alternative to task branches or as a starting point to evolve to simpler and more efficient practices.

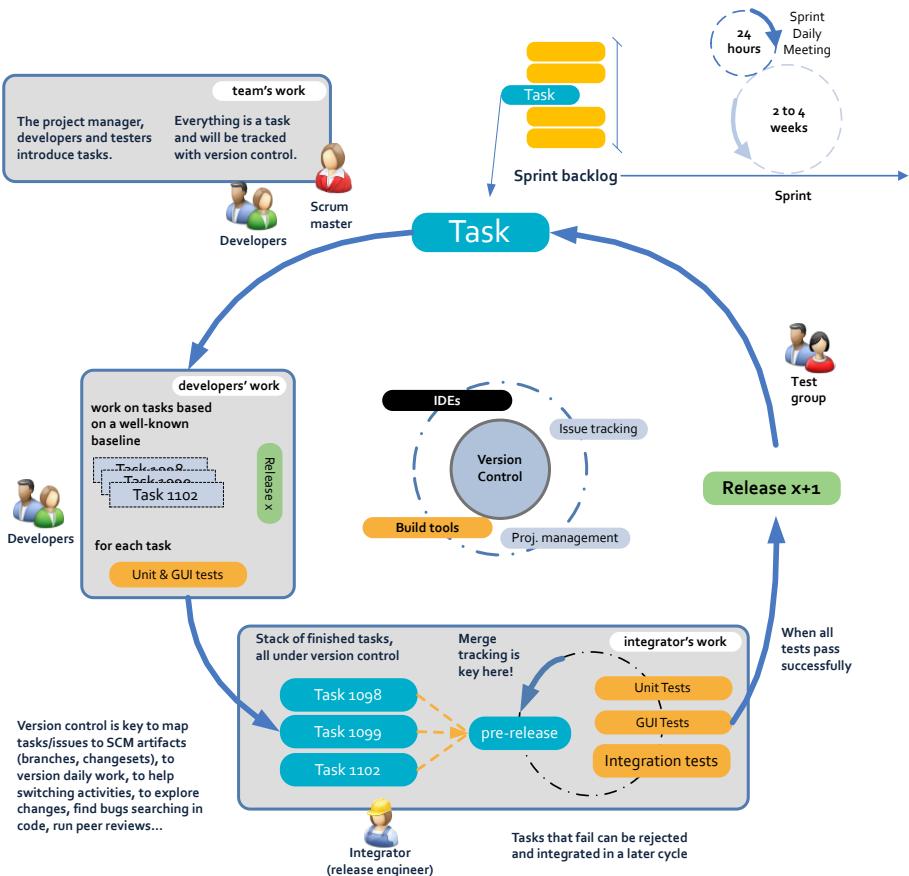
Branch per task with human integrator

Cycle definition

Consider this like a prequel to the "Branch per task pattern" I explained at the beginning. Everything is about one task per issue in the issue tracker, tasks being independent, short, checkin often and so on holds true. The only difference with what we saw there is that in this variation the integration phase, the merge, is performed by an actual team member instead of by a mergebot or CI system.

We used to work this way. In fact, we followed this pattern for years with a great degree of success. Task branches combined with an integrator in charge of merging every branch back to main or the release branch at hand. We only moved past it when we finally automated the merge process with mergebots.

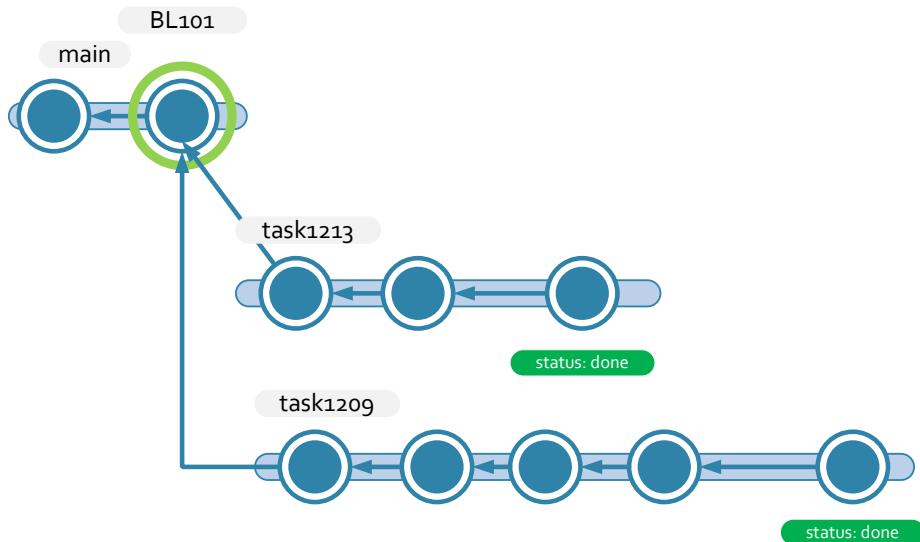
I think I've drawn the following graphic a hundred times over the years and explained it in many different places from meetups and product demos to classrooms during my days as a university professor.



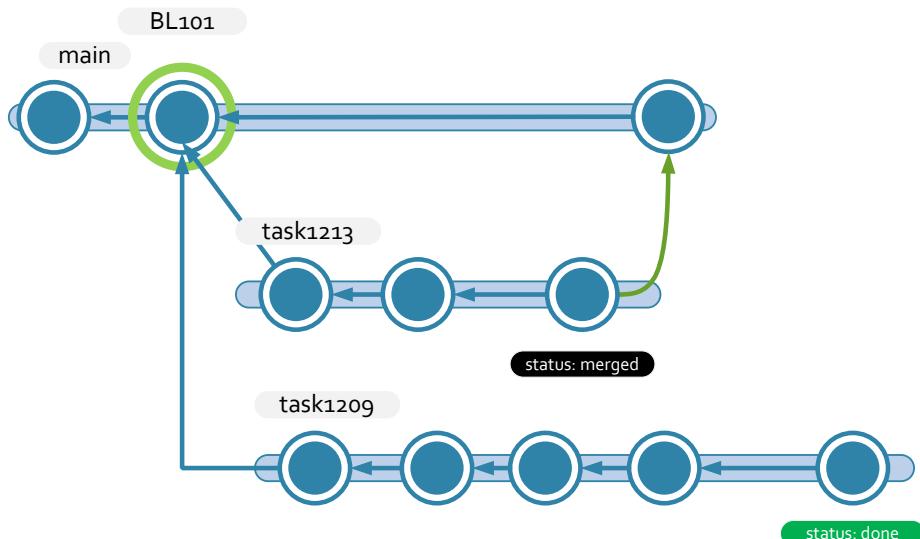
- It all starts with a task, normally coming from the sprint backlog if you are using scrum, but it can be taken from a Kanban board or any other project management mechanism you use. It is important to note, as I described in the chapter about task branches, how every single task branch has a one-to-one relationship with an entry in the issue tracker. Depending on how prescriptive and hierarchical your team is, everyone will enter tasks or only some designated members.
- Every developer takes one branch and works on it and marks it as done when their finished. Tasks are tested individually. It can happen thanks to a CI system (recommended) or because of tests manually triggered by developers.
- Then, at a certain point, a stack of finished tasks exists. Of course, the smaller the stack is, the fastest your team will move. And here is where the integrator enters into the scene. The integrator takes the list of pending branches to be merged and merges them, resolving any possible manual conflicts thanks to their project experience or asking the involved developers to help. Fortunately, the percentage of manual conflicts is small and conflict resolution is normally the smaller of the work efforts in the integrator's bag.
- Once the branches are merged, the integrator creates a "pre-release" and launches the test suite (with some ad-hoc batch system or with the help of a continuous integration software).
- When all tests pass and a new version is created and labeled, the developers will use it as starting point for the new tasks they'll work on.

How it looks like in practice

The day of the integrator typically starts with a list of tasks to be merged. The integrator takes them from the issue tracker (list of finished and not released tasks) or through a query to Plastic SCM if branches are tagged with their corresponding status.



In the figure, the integrator has to take care of merging **task1213** and **task1209**. They can start with **task1213** and merge it to **main**. In this case, there's no possible conflict between **main** and **task1213**.

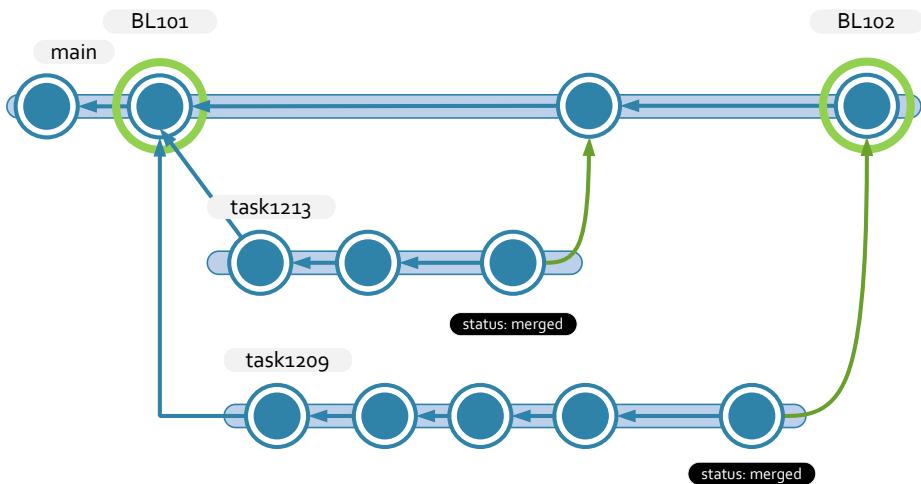


Prior to actually confirming the merge (checkin the result of the merge from his workspace), the integrator will normally build and test the project. This really depends on how fast tests are. In our case, we used to run the test suite manually after every task branch was merged to **main** and before checkin. This was before having tenths of hours of automated tests, when the suite just ran in about 5-10 minutes.

Of course, having a human integrator doesn't mean they have to trigger tests manually. A CI system connected to the repo could have tested [task1213](#) and [task1209](#) so these two branches would only be eligible if the tests pass. A caveat, though, and we experienced this ourselves, is that those branches are typically tested in isolation instead of merged with [main](#). And, if you already automate that step, then the manual integrator is rendered obsolete (as happened to us) because there's no real reason not to let it merge back to [main](#) too.

Anyway, in our human integrator scenario, the integrator checkins the merge from [task1213](#) to main once they consider it is good enough. Notice the integrator doesn't label the changeset at this point because another task is queued to be merged. Since the process is manually driven, grouping tasks makes a lot of sense (and it is indeed, a good reason to automate the entire process and avoid turning new versions into events that can slow down the whole cycle).

Then the integrator decides to merge [task1209](#), run tests, checkin, and if everything goes fine, label [BL102](#), which will be used as the next stable starting point for the developers as well as being published and/or deployed to production.



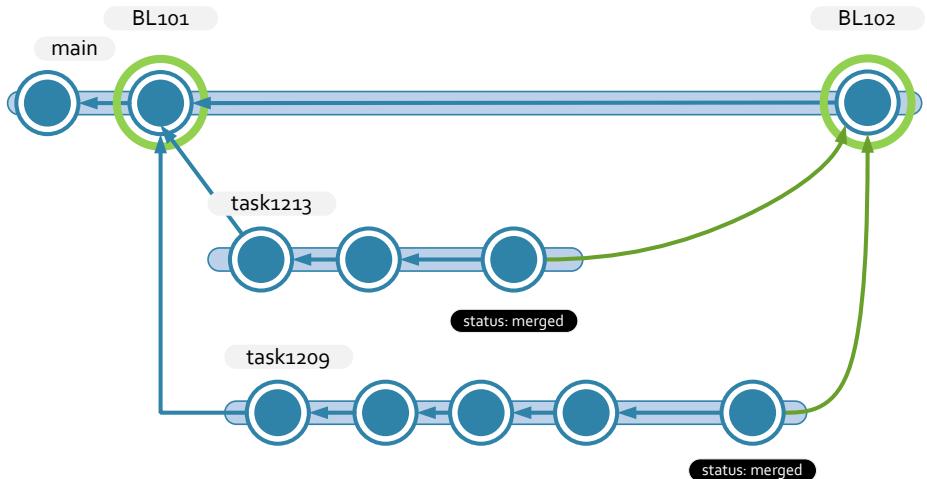
When I say the new [BL102](#) will be used in the next iteration, I don't mean developers were stopped while the release was created. They could safely continue working on existing tasks, or create new ones starting from [BL101](#), since [BL102](#) didn't exist yet.

The branches started from [BL101](#) don't need to be rebased to [BL102](#) unless they are in conflict, and since there is a human integrator, this merge is normally not requested because the integrator will solve the conflicts during the merge phase.

While merging [task1209](#), the integrator might have found conflicts that required manual intervention. In small teams, the integrator can be a rotating role among experienced developers, so they'll be able to handle most of the merge conflicts. If not, if the integrator doesn't really know how to solve a given conflict, they'll ask the author of the changes to reconcile the changes correctly.

The integrator decides to checkin after the merge of each task branch but in Plastic they are not forced to work this way (as they are in Git, for instance).

An alternative merge scenario would be as follows, where both [task1213](#) and [task1209](#) are merged into the same changeset.



In our experience, the disadvantage of doing this with manual merges is that you can be forced to undo more merges than you would like to. Suppose you have ten branches to merge. You have been building and launching tests manually for each of the first six. Then, the tests of the seventh fail after it was merged. If you undo changes now, you'll undo the six previous merges. While the result looks cleaner because there is a single changeset for all the branches, it can well be a great inconvenience if something fails.

Pros & cons

Having a human integrator has several pros and cons.

Pros	Cons
There is always someone in control of all merges, so there are few chances someone merges quick and dirty because they are in a hurry and need to leave.	It doesn't scale. I don't mean this is only for small teams, I've seen it happen in large ones too. The problem is that the cycle you can implement is not fast enough to achieve true DevOps. Every single manual step slows down the whole project.
Most team members don't need to merge on a daily basis, which greatly simplifies onboarding. Developers without a solid understanding of merge can easily work just doing checkins to their branches, not worried about how they'll be merged.	Creating a new release becomes an event. "Hey Ms Integrator, please wait for task1211 because we do need it in the next version". Sound familiar? If the process is automatic, there is no need to wait. You'll have a new version today, and a new one tomorrow (or maybe even a bunch of them), so there is no need to wait.

With a fully automated cycle driven by the CI system or a mergebot, you get a scalable process where releases happen on a continuous basis.

The only downside I can think of is that every developer will need to eventually merge down some changes to solve a conflict, something they were shielded from with a human integrator. And, of course, since everybody can merge, there's not the same level of confidence that a wise integrator will always

get the merges right. But, good test suites seem to handle that pretty well. The world has moved towards automation for a reason.

When to use it

Having a good branch per task cycle with a manual integrator is a good first step towards full continuous delivery and DevOps. I wouldn't use it as a final goal but as an intermediate step or starting point.

If you are not using any sort of branching pattern and experience frequent broken builds and regressions, this workflow will certainly help you move forward if you don't have enough tests or infrastructure for a fully automated alternative.

Mainline only for unmergeable files

Mainline is when an entire team directly checkins to a single branch, typically main.

It is easy to confuse this with trunk-based development, although I believe they are radically different.

Most teams doing trunk-based nowadays don't checkin directly to main. They use proper task branches if they are in Plastic, or they commit locally then push if they are on Git. In this second alternative, their local branches are their real working branch (the equivalent to the task branch in Plastic, although less organized, in my opinion) and they never checkin to the master on the remote site simply because Git doesn't allow that by design.

That being said, I'd only use mainline to work with files that need to be locked because they can't be merged as we saw in "[Locking: avoiding merges for unmergeable files](#)".

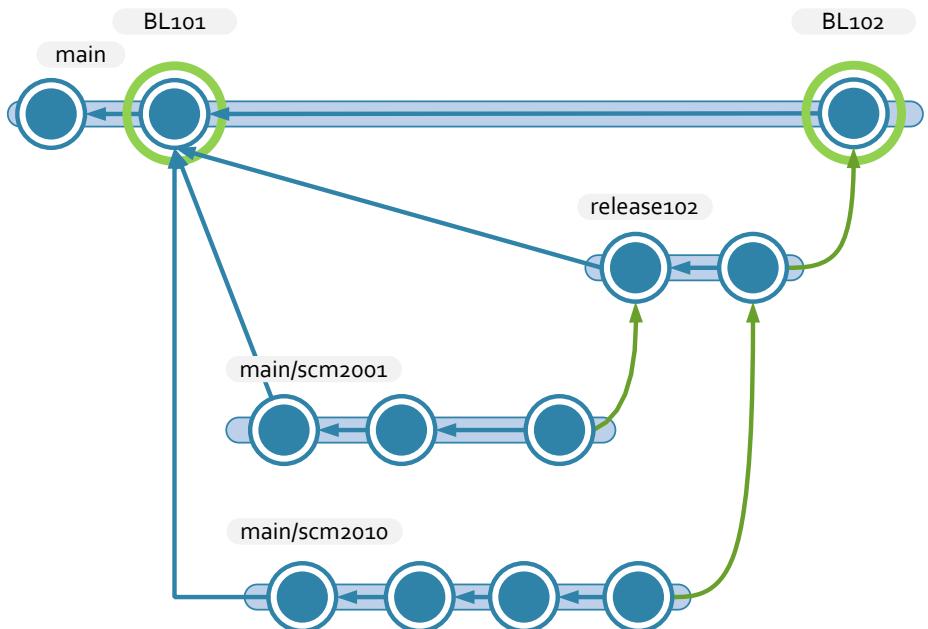
Task branches don't impose a big burden, come with lots of flexibility and are way better than mainline in all circumstances except when locking is a must.

Release branches

All through the book, I used the main branch as the integration spot for the task branches. But, we often find teams where they use release branches to stabilize the new versions before actually releasing, or to simply have a spot to safely and quietly perform the merges before moving to `main`.

A typical scenario with release branches is as follows:

- Task branches start from the stable release, typically labeled on `main`.
- But they are merged to a release branch instead of directly to `main`.
- This allows the team to keep `main` clean.
- And releases to progress and take its time.



We used release branches ourselves very often before we moved to full automation mode.

The release branch was useful for the human integrator because:

- Creating a release took some time, so if something super urgent happened, main would be clean to start a different one with the hotfix in parallel.
- In the event of things going wrong with tests or the code itself or last-minute fixes, the integrator could apply small changes directly to the release branch without passing through the entire cycle. This is of course a small cheat, caused by the fact that manual releases took time.

I normally don't recommend this model anymore if you implement a fully automated cycle for many reasons:

- It is easier to automate a simple workflow where task branches are simply merged to main, without creating intermediate steps. Here laziness matters ☺ but also keeping things simple.
- If new releases can be created continuously, one per each task branch, it doesn't make a lot of sense to create an intermediate new integration branch too. I mean, release branches make sense when you group tasks together, but automation renders that irrelevant, because every single task can be merged, tested and released. Hence, with automation, these intermediate steps don't make much sense anymore.

They can be very useful, though, if you can't achieve full automation for whatever reason, if you need to group branches to be tested because tests are too slow, or even in more complex cases where several releases need to be created in parallel from the same codebase.

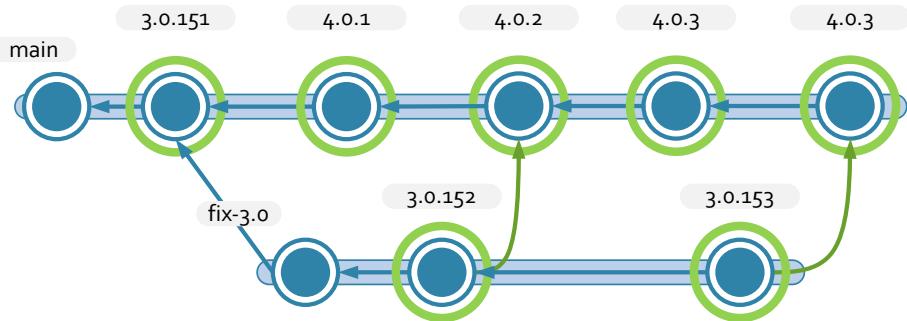
Maintenance branches

What is the best way to maintain older versions of your product while you evolve a newer one? Branching can definitely help here, but there is a big warning to keep in mind: the actual limitation to

handle parallel versions is not the version control but the actual amount of work your team can handle. Dealing with parallel versions is extremely time consuming and can expand to use all the available resources until nothing is left to produce new meaningful work.

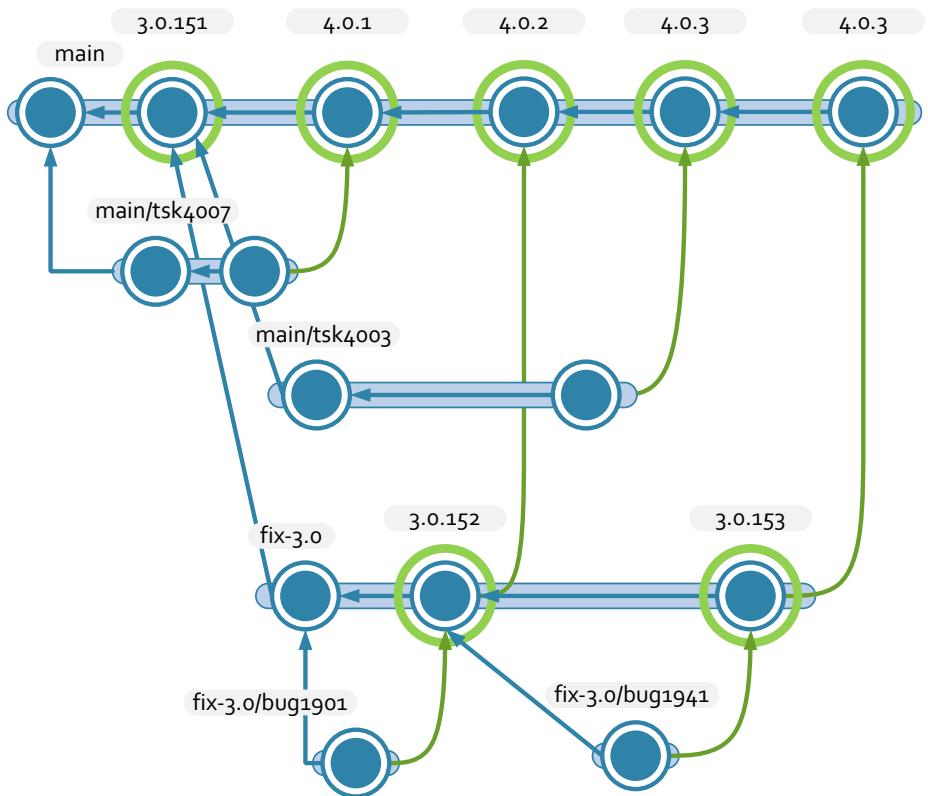
A typical maintenance layout

Here is the typical layout and evolution of a maintenance branch:



- The `main` branch was version 3 until version `3.0.151`.
- At this point, a new branch `fix-3.0` was created, starting from `3.0.151`.
- From there, `main` continued evolving, this time turning to `4.0`. New versions `4.0.1`, `4.0.2`, etc. are created later. Most likely, some of the initial `4.0.x` versions were not yet public to everyone, or even not public at all, but they are created frequently as the result of the development iterations, exercising the entire DevOps cycle.
- In parallel, `3.0` continues evolving, normally only with bug fixes. This way `3.0.152` and `3.0.153` are created and deployed to production or delivered to customers depending on the nature of your software.
- Note how the new `3.0` versions are always merged to `main`. This is because a bug found on a maintenance branch normally affects the development version too, so it needs to be applied.

Of course, the Branch Explorer diagram above is a simplification that only shows the primary branches. A more realistic view would be something like the following diagram shows: short lived branches all around being merged to `3.0` or to `main` depending on whether they apply to one or the other line of development.



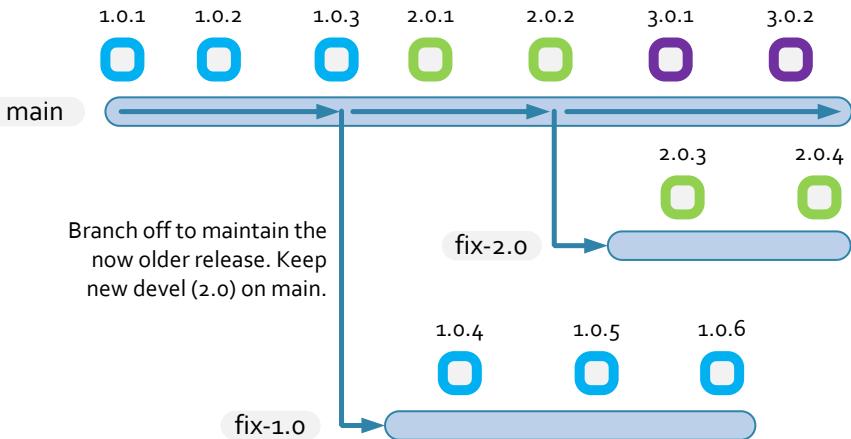
Simple proposal for maintenance branches

This is how I summarize our proposal to deal with maintenance releases:

- Keep the evolution on main. There are other alternatives such as evolving the new **4.0** on a dedicated branch and merge it to main only when it is stable enough, and then swap lines there, creating the **3.0** maintenance branch from the last changeset before the **4.0** merge. We have done that ourselves several times over the years but honestly, I consider it an overdesign now. It is much simpler to reduce the number of branches to keep it simple.
- Keep maintenance on **fix** branches. Branch off when you decide from this point onwards that main is already going to be the next version. It can be the moment before activating an important feature, a big UI change, etc.

The following figure summarizes the proposal:

Latest release evolves on main

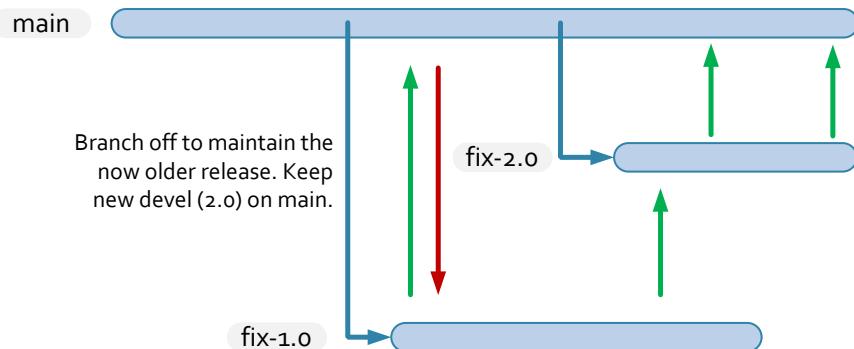


When to merge maintenance branches back

There is a really simple rule of thumb:

- You can merge `fix-xxx` back to `main` as often as you need to apply new bug fixes or improvements.
- But you can't ever merge `main` to `fix-xxx` or you will convert `fix-xxx` in your new version, and that's something you don't want to do.

This might sound obvious, but I want to make it crystal clear.



You never merge down from `main` to `fix-1.0`, never from `main` to `fix-2.0` and never from `fix-2.0` to `fix-1.0`. Otherwise, you won't be turning 1.0 into 2.0, or 2.0 into 3.0, etc.

Now, the merge upwards from older to newer releases is safe and in fact should happen often.

One of the challenges you find when you have several open parallel lines of work is when to perform these merges.

Our internal rule is:

- Treat `fix-xxx` branches as task branches. Whenever there is something new on `fix-xxx`, merge it to its parent branch.

- This way we eliminate the risk of missing an important fix made on an older release.

Not sure if I really made it clear. The alternative to this "merge immediately" might be: remember to merge (or automate it using your CI) the `fix-xx` into its parent the next time you create a new release on the parent. Ok, I'll try again with the example above: whenever you make a new version in `main`, remember to check if there are pending merges from `fix-2.0`. This alternative mainly works for scenarios with human integrators, because if your process is fully automated, there's no really a need to delay the merge, and as soon a new version is ready in `fix-2.0` it should trigger a new one in `main`.

As you can see, in our example, a new version labeled in `fix-1.0` will trigger a cascade of new versions in `fix-2.0` and `main`. That's pretty normal because it can be a bugfix in code still shared by the three versions.

Problems with maintenance branches

As I said at the beginning of the section, maintenance branches are problematic. And I don't mean they are from a branching perspective; Plastic will treat them as it would do with any other branch, but from an organizational level.

Maintenance branches were a hot topic in desktop software, because new versions were released only a few times a year or even every few years, and upgrading had a licensing and business impact.

Nowadays, the cloud and SaaS are a reality for a vast majority of teams, and the branching strategies were greatly simplified in that sense. You don't really expect to use the previous version of a hosted service, but to seamlessly use whatever the latest version is as soon as you keep paying the subscription.

Under that premise, it is less and less worthwhile to keep maintaining version `3.0` when `4.0` is already in production, since there's no business need for that.

In the end, the entire DevOps philosophy is all about achieving a fast pace of small changes that go immediately to production, so the big jumps between versions become something of the past. It is more about a single `main` branch evolving through time than about taking care of parallel versions.

That being said, there are a few things to keep in mind:

- Not all teams develop SaaS cloud hosted software, so for many of you maintenance releases can still be a big thing.
- Even in Cloud, sometimes it is important to rollout two different versions to production, so the new one is deployed only to a designated group of users to validate it, while the bulk of users continue with the older one. If the test takes a few weeks to complete, it is very likely that you need to support parallel evolution in the two versions. Of course, a maintenance branch for the old one will help. But the good news is that it is very likely that this maintenance branch will be extremely short lived (a few weeks at most) and you'll almost never have to deal with more than two parallel lines of development.

The big problem with maintenance branches is when they start to pile up. I think it is no longer a big issue for most teams, but some enterprise software and server and desktop applications can still be impacted.

I mean, what happens if every few months you launch a new major version due to business requirements and then you find yourself having to maintain `3.0`, `4.0` and `5.0` while evolving the upcoming `6.0`?

The problem, as I repeatedly said, is not that much the branching and merging themselves, but the fact that your team will have split its focus to develop in four different lines of work (unless you have a

dedicated team for each version, which in my experience is a nice rarity that almost nobody benefits from).

This split in focus means dealing with continuous context switches ([3.0](#) might be arcane and odd by the time [6.0](#) is under development) and a huge negative impact in productivity.

I had the chance to visit teams where they had the impression (backed by numbers) that they were unable to get anything new done because they were constantly fighting too many open fronts maintaining old releases.

The solution doesn't come from a mere technical alternative but from a strategic decision: try to minimize the number of open versions. Yes, this is much easier said than done, but if you are the CTO or technical lead, you should really strive for this simplification. Does the business really benefit from not upgrading customers to newer versions if they don't pay the upgrade? Can't sales simplify the process? Maybe target for more aggressive discounts to ensure everyone benefits from the newest and better versions instead of being stuck with an old dinosaur?

Even in the most organized and test-aware teams, a version from five years ago will be lacking. Probably, the old test suite is now harder to run because of some infrastructure changes (the old tests where not ready to run in the cloud while the new ones can), or the old version has far more tests than the new one, etc. So, touching the now ancient version is a nightmare for everyone. It doesn't mean branching it is harder than creating a task branch on the current version. Plastic doesn't care about that; it is about the actual cost for your team.

The ideal is, of course, to have a single running version and really minimize bugfixes on the old ones as much as possible (to zero if you can afford it). Probably, during the first few days of the new version being public you still have to do some fixes or small improvements in the previous, but this is a temporary situation and the old one will be naturally freezed in a few weeks.

Of course, there is always an equilibrium: it is better for the dev team to immediately mark old versions as obsolete when you launch a new one, but forcing all customers to upgrade might be a nightmare for them sometimes unless you ensure the transition is super smooth. Sometimes, you have a new version that is much better, with a greatly improved UI and many new features, but the cost of change for some customers is big, and they really prefer not to invest in the change. Telling them "no" and forcing them to upgrade might be good from a programmer perspective, but it would mean not really putting yourselves in the customers shoes which is never a good choice.

If that's the case, you'd be forced to deal with the older release, unless you really plan this in advance and ensure that upgrades roll out seamlessly and frequently to prevent big jumps down the road.

As you can see, the challenges of maintenance releases goes far beyond branches and merges and well into product and project management, sales and overall business vision.

Where to apply bugfixes

Suppose that, for whatever reason mentioned above, you find yourself with two or more maintenance branches open in parallel plus the main development.

What happens when a new bug shows up?

The team must find which is the older version affected and apply the fix there.

Yes, it might sound crazy and extremely time consuming if you have many parallel open versions, but that's why exactly you should try to minimize that number.

My advice here is:

- Locate the older open version affected by the bug (I stress the importance of "open". It is not about being a software archeologist and finding that the bug affects a version from 10 years ago and branch from there if that version is no longer supported).
- Create a task branch from the latest on that maintenance branch.
- Code and test the bug fix.
- Merge it to the maintenance branch following the regular procedure, passing tests, etc.
- Cascade merges up until main, creating new versions of each maintenance branch.

Then, it is up to you and your business needs to decide if you release all those versions; but, from a technical perspective, you are giving a solution to the business folks and you are covered. You give the business the right tools, now it is a matter of using them wisely. The opposite, creating the new releases only if someone complains, is a worse choice if your process is fully automated. If it is manual, then saving time must be worth it.

Please note I'm talking about merging changes, not doing cherry picks.

I once visited a team developing a huge set of products based on a shared codebase and they had a complex procedure and set of tools to cherry pick bug fixes among the open versions they had. It was extremely time consuming and complex and hard to keep up to date and to really understand what was going on. Try to avoid that path at all costs.

How we do it with Plastic SCM

Since we develop on-premises software (we have a Cloud version too, but you can setup your own server so the upgrades are outside our control) involving servers and desktop apps, we were traditionally hit by the problem of maintaining old versions.

It was very common to try to group new features and only make them visible when the new big release came out. This was a practice we all saw promoted by the big software companies for decades and somehow, we all had considered to be a good way of working.

It had a few big problems:

- First, we could be working for months on new things nobody was seeing. It could be November and new teams evaluating Plastic were downloading the "stable version" from February, missing all the new improvements made during the interval. Sometimes we were shocked to see how "old" the version a customer had since we were all used to the new features.
- Constantly splitting time between doing releases for stable and new software.

The solution was to simply create frequent public releases on a weekly basis with both fixes and new features, and roll out new things during the year instead of waiting to group them all together.

Customers are not forced to upgrade every week, of course, so they can choose their own pace.

We don't have to worry anymore about not making things public to have a "big release with tons of changes" at a given date.

Marketing and sales still like to see major versions coming. So, what we do is as follows:

- Every January (sometimes it can be delayed a little bit) we change major version. We go from 6 to 7, 7 to 8, etc.
- The early 7.0 is exactly the same as the latest 6.0, well, adding the small fixes and improvements that go into every release.

- Over the year the new 7.0 continuous evolves while 6.0 is frozen.
- At the end of the year, the latest 7.0 is very different and much better than the one at the beginning of the year, but the changes were rolled out progressively.
- We lose the "surprise effect" of packaging a lot of new things together.
- But overall it is better for everyone. Newcomers have access to the best possible version (so we don't lose someone because they can't see a feature that is waiting in the fridge), and existing customers see a more gradual evolution
- Feature toggles help us ensure risky changes don't have a negative impact and let us test those changes internally or with selected customers.

Conclusion

- Try to minimize the number of parallel open versions. If your customers are fine with just having a single version that receives updates continuously, everything will be better than trying to keep 3.0, 4.0 and 5.0 running in parallel.
- If you really need to handle maintenance branches, keep them as simple as possible and cascade up changes creating new versions.
- Contact us to double check your setup. We have seen many teams over the years and very likely we'll be able to help you with our experience. We'll do our best to help ☺

Branching for product variations

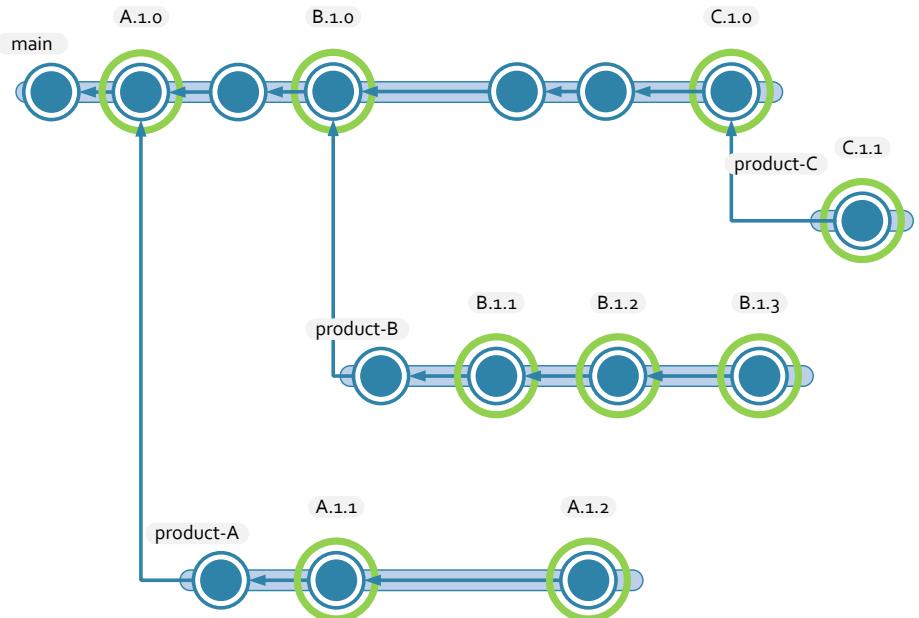
There is a recurring request that shows up repeatedly from some teams adopting Plastic and it is their intention to handle product variations with version control. They think branching is a natural way to handle product variations and customizations.

Normally, they have a solid business based on selling custom software to their enterprise customers. Each customer has slightly different requirements, so while there is a common base that is used as starting point, new development needs to be done to win the deal. And, the development normally takes a few months of work.

Product variations can certainly be handled with separate branches, but since I'm trying to be prescriptive to be here goes my advice: **don't use branches for product variations, use feature flags instead.**

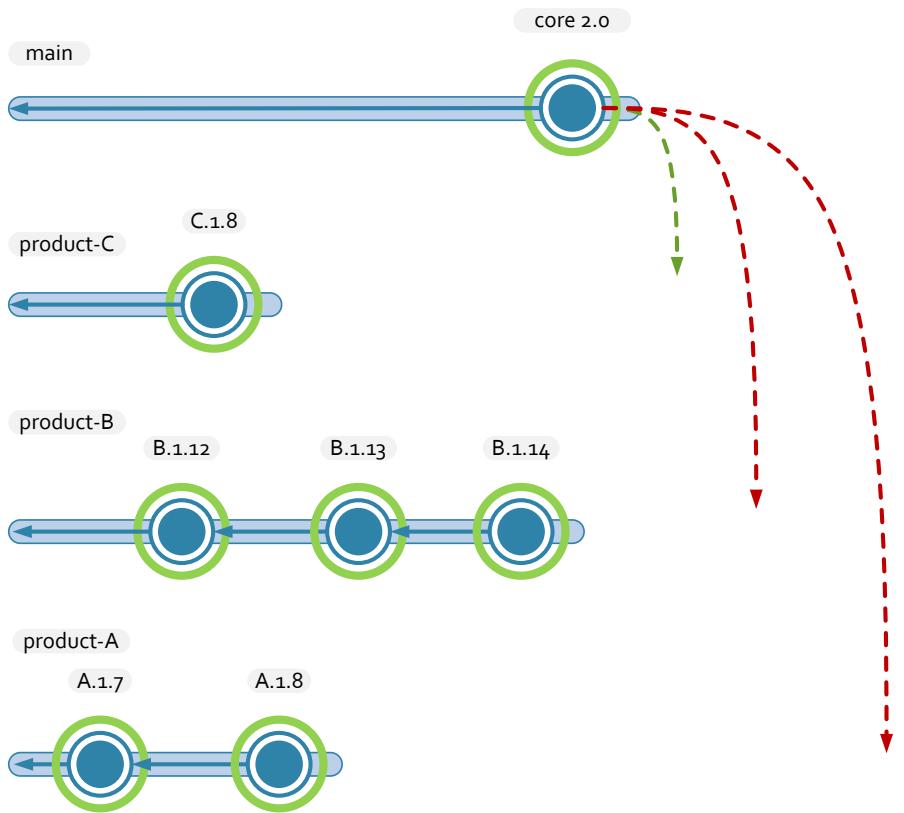
The problem with many open development lines is quite similar to the one having many open maintenance versions: they tend to drain all available time. You end up spending more time applying changes here and there than doing new development.

The typical scenario starts as follows:



- Development happened on `main` until a product is ready to be released to the customer. Product A version `A.1.0` is labeled and then further evolution and maintenance happens on `product-A` branch. In my experience, the beginnings are never that clear. If we have to import an old SVN repo, the initial branch most likely didn't happen in such a clear way both probably well after A had evolution and close to the point where product B needs to be released. For the sake of simplicity, I'll keep it in this "close to ideal" form.
- Once `product-A` is out and used in production by the customer, further development happens both in `main` and `product-A`. Normally, `main` will receive a great evolution to get ready for the upcoming `product-B`, while `product-A` only receives minor fixes. Unfortunately, very often `product-A` receives major new features to respond to customer needs.
- `product-B` reaches its 1.0 version and is branched off.
- Then, the story repeats for the new `product-C`.
- At this point, the team is actively working on four major branches. If they are lucky, `product-A` and `product-B` won't be much trouble and they'll be able to focus on C and the upcoming contract or major core upgrades. If they are not, they'll be constantly switching projects.

What I described so far is somewhat the starting point. But later, what the team will see are a bunch of parallel lines without any confluence at sight.



As the figure shows, the team is now dealing with four parallel lines of development. And now consider a major improvement was done on `main` and the person responsible for each product (typically some account manager with technical background) wants these updates to secure a renewal or to renegotiate a contract. How can you apply the changes?

- In this example, we can merge from `main` down to `product-C` since it seems we didn't create a new `product-D` yet, so the evolution might well be applied to it.
- But we can't merge (red lines in the figure) down to `product-B` and `product-A`! Why? Because then, we'd turn them in `product-C`, which normally is something teams don't want to, very often due to stability concerns.
- If bringing changes made on `main` to the other product lines was not an issue, then frequent merges should happen frequently to push changes down to the different product lines, but that's normally not the case. In fact, if that was the case, why would you have three different product branches?

Since merge is normally not permitted due to the aforementioned issues, teams tend to start doing cherry picks, making maintenance even worse and more difficult to track. Cherry picks, as described in the merge chapter, don't create merge tracking, which won't simplify later merges.

Typical challenges found in setups with multiple product branches

Here is a list of typical problems and challenges found on teams with strategies based on branches per product:

- Somehow, they have a core that all products reuse. The core is typically developed on main. But in reality, this core is not that cleanly defined and trying to componentize the project becomes an unachievable goal the team has been pursuing for years.
- In some situations, we even found that main is not even stable (sometimes it doesn't even build!) and stabilization is done on the product branches. This is one of the reasons why trying to unify the different products is repeatedly discarded, since account managers are very scared of regressions and the negative impact on their customer.
- Fixes are done ad-hoc on the product where they are found, and often the same fix is done more than once by different developers, with the associated wasted effort. Teams dream of a magical solution to be able to identify and apply fixes correctly and avoid working twice.
- Some teams even implemented some home-grown patch system to be able to track which patches were applied to which products. Since merging between branches became a nightmare when not totally forbidden, they need to do patch-based management.
- Teams are usually stuck with some ancient version control, like SVN or even CVS, which makes merging even worse. And somehow, they expect a new version control to fit like a glove in their environment and magically fix all their issues, which is not going to happen because their problem is deeply rooted in their way of working, not only the version control.
- Normally, the lack of automated tests adds up to the pile of problems, making every possible move or refactor almost impossible.



Game studios reusing an engine are not a valid example of this pattern. Different titles are sufficiently separated from each other to deserve their own evolution, and the engine is usually well defined and handled by a core team of experts. In fact, it is more common to find each game on its own repo instead of just as different branches. All game teams might do changes in the engine, but they do that on a branch of the engine repo, and those changes are merged and reviewed by the engine core team. Particular changes in the game itself, or its entire evolution indeed, is separated enough not to be reused between teams.

Proposed solution

When we hit a situation with multiple product branches, we insist on the importance of having a standardized product without variations. If variations are a must due to business needs, then we make it clear they should implement it through feature toggles on a single codebase instead of having multiple branches.

These statements are normally shocking for the teams who expected us to come up with a magical solution that simply merged all their changes and fixes in a better way.

A story of penicillin and superheroes

In these situations, where the solution is not a new shiny super advanced technology, but simply working in a different way and using solid well-known tools in a better way, it always brings to my mind a comic that I read when I was a kid.

I love superhero comics, but I think I never re-read this one, and somehow, I prefer to keep a blurred and probably more beautiful idea.

I think it was in some Crisis in the Infinite Earths comic. The hero wanted to save a loved one from a certain death due to a rare disease. Then, he arranged to travel to the future and after much trouble he was able to meet with some doctor.

Then it comes to the surprise.

The magic cure for all the diseases in one thousand years was... penicillin. The hero was shocked. Just penicillin?

Humans had evolved in such a way that this basic antibiotic was all what they needed. They were simply different, they did things differently, and many diseases simply had vanished over the centuries.

The hero was discouraged and speechless.

The solution was not a magic pill but doing things in such a different way that the most horrible problems did not need a unique cure but simply to be avoided.

Our proposed recipe is:

- Start implementing automated tests. No matter how hard it sounds, it is possible and it will be a cultural change so profound that everything will change after that. Of course, you won't have full coverage in a few days, but little by little, in a year from now you'll have a solid safety net.
- Start doing peer reviews in code.
- Work on short task branches instead of on huge feature branches.
- The main goal is to ensure main is stable; then, that you can ensure stability in all product branches just by running tests.
- In parallel: you need to talk to the business people so that they start preparing for a standard version. We are going to fight against totally customized versions for each customer. This is simply the old developer versus salesperson fight of "don't say yes to everything" and trying to find the right balance.
- Teams normally have an idea (and dream) for standardization of their product, sometimes even some work in progress.
- Implement customizations as configuration: it can be through config files and feature toggles, or through conditional compilation. Anything is better than fighting lots of parallel branches that will never be merged.
- Then we move them to task branches + trunk. A single line of development (trunk) with short lived (1 to 10 hour) task branches.

If they achieve it, teams will go from fighting four or more parallel branches to simply focus on the development of small incremental steps that are fully tested. This doesn't mean they'll be deploying to customers every day if their business doesn't want that, but they'll have the flexibility and the tools to do

it, and to be always ready to release.

Once they move to the new paradigm, step by step as stated above, they won't have more cherry picks, no more figuring out which products are affected by what, no more how to apply these core changes to the four product lines. All of that will simply happen auto-magically 😊

No automated tests

Well, this is not an alternative branching strategy but a very common problem for many teams. So, I thought it would be a good idea to introduce here a possible path for solution.

Many teams learn about task branches and fall in love with the idea, only to get demotivated by the fact they don't have any automated testing and hence they can't implement it.

What can you do if this is your case?

- You can implement "[Branch per task with human integrator](#)" from day one, but it will only work if you really start implementing automatic tests.
- The manual integrator will help you implement the concept of short tasks, and evangelize the whole team on its benefits.
- Every task will be code reviewed (no reason not to do it), which will unlock collective code ownership. Of course, you'll catch bugs and establish a common interest in code quality. The senior members will also continuously train the less experienced developers.
- You'll interiorize the concept of frequent releases, short tasks, code reviews, and stable starting points for new tasks.
- You can implement some sort of validation for each task, and also manual validation of the new versions to avoid regressions.

Of course, this is a compromise to get you started. None of this will hold together if you don't start building a solid foundation through automated tests. You don't truly believe in task branches if you don't believe in automated testing.

To get started with tests, check the section "[Automated tests passing on each task branch](#)".

Slow builds and slow tests

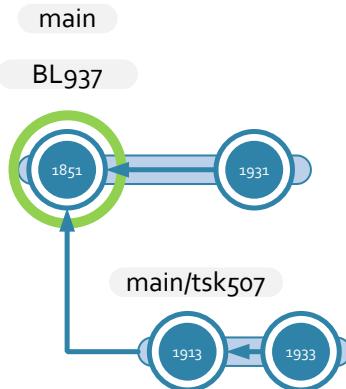
By slow builds I mean both projects that take hours to build, and test suites that take hours to run. In some industries, especially in long life projects based on C++, it is not uncommon to find that rebuilding the project can take 6 to 10 hours or even more. Normally, by that they mean a full rebuild, which is not that common since most of the intermediate compilation objects are reused, but even a faster build in a CI system can take a few hours.

These slow builds greatly complicate the pure task branch cycle where every task is expected to pass the full test suite to ensure it is ready for production.

This is the solution we propose:

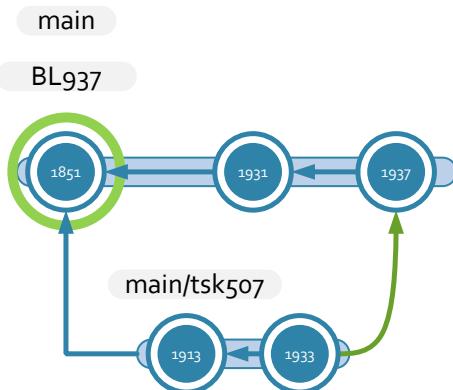
- Stick to task branches and testing each branch. Since each branch can't afford the, let's say, 10 hours of build and test, select a faster and smaller subset. It is always possible to select a minimum test suite that can be run in about one hour. If the problem is just the build that takes too long, the CI system can be setup in such a way that it reuses most of the intermediate objects or takes advantage

of a build accelerator.



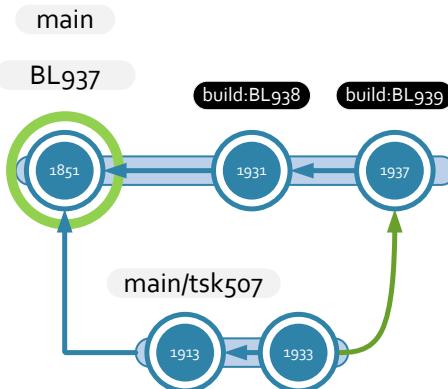
A full build would take 10 hours, so we can't afford that.
Pass a subset that takes 1 hour at most

- Once the lighter test suite passes, the branch `tsk507` is merged to `main`. It can be used as a starting point for other task branches since it is "good enough" as it passed the minimum test suite. Alternatively, you can stick to only allowing branch creation from the last stable build, `BL937` in this case.



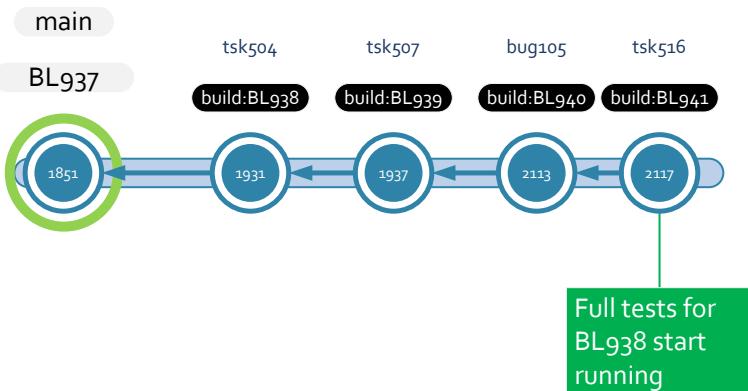
The task is merged with a smaller suite.
The new changeset 1937 is good enough
to be used as the base for new branches.

- We prefer not to create a label for each of these "candidate versions" if possible, and we simply set an attribute "build" with the number of the current build. Every changeset on main is a valid "release candidate".

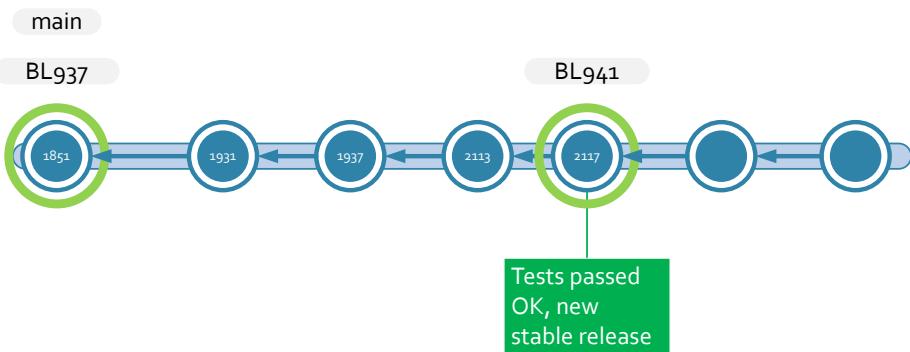


A new "version" (build) is created after each task is merged. We like to mark it with an attribute even if labels are reserved for stable releases only.

- After a few more task and bug fixes, branches are merged, the `main` branch will look as the next figure shows. At this point, we have a group of four branches that can be used as the basis for a new public release. Ideally, the smaller the group is, the better the chance is to achieve agility. But, if builds take too long, then maybe it is good to have a group of five to ten branches. In our particular case, our mergebot triggers the build with four branches or if there are no branches waiting to be merged (which usually happens after working hours, which ensures the mergebot uses its time wisely at night).



- If the entire build and tests pass, then the changeset 2117 that was used to group the four branches will be labeled accordingly as the figure shows. Of course, meanwhile the project continued running and now there are new changesets on main waiting to be released.



- In case the full build failed, the changeset 2117 wouldn't be labeled. The team will receive a notification and the branch will enter "broken build status". The priority of the team will be to put a new changeset on main with a fix for the test or tests that were broken. We don't recommend that you go back (at least not usually) but to simply create another task to fix the issue. It doesn't mean we don't strongly recommend root cause analysis to determine why something broke and whether a new test should be in the "quick test" phase for each task to prevent these kinds of errors to reach release tests.

This approach tries to get a valid solution for the slow build / slow test suite problem. New "release candidates" are created on a continuous basis and each of them is eligible to be converted in a public release if it passes all tests.

The long-term goal is reducing build and test times and getting rid of this second stage, and sticking to the "every single changeset on main is a valid release that can be deployed" mantra. But, at least, while you work on this and you keep it as an important goal, you can benefit from the improvements of task branches and trunk devel.

Big Bang integration

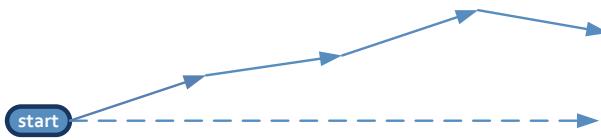
Big Bang integrations are the root of all evil in software development. So, while I think it is good to explain them here so you can all avoid them at all costs, I sort of fear I'm unchaining some wild sorcery.

(I have to thank my good friend and mentor Wim Peeters for the original graphics I'm going to display here. Back in 2006, only a few months after we started Plastic SCM, we had a local presentation where we introduced the product to companies nearby, long before it was even ready for production. Wim was so kind to travel to Spain and share his knowledge about software configuration management.)

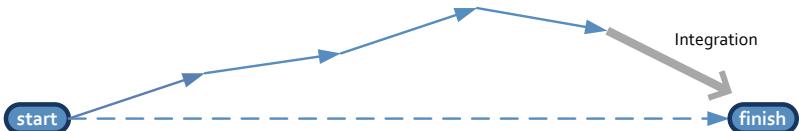
It all starts with an individual or entire team making contributions to a code base. They work in isolation and the rest of the project doesn't get their changes. It starts looking like this:



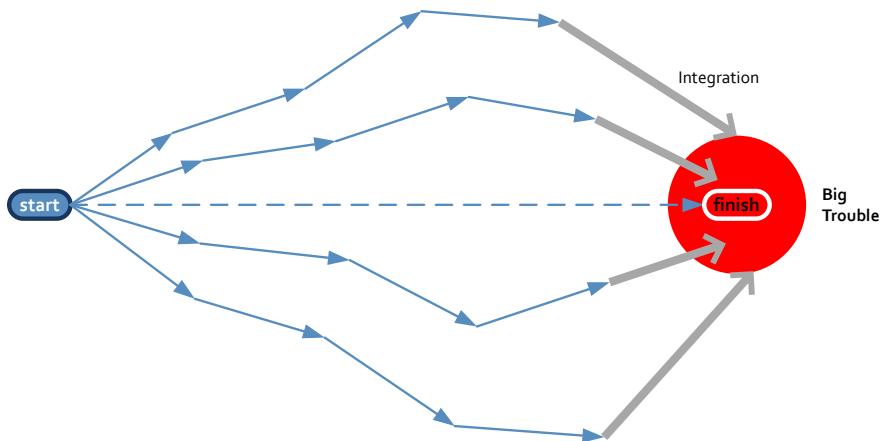
Then as they continue evolving, more and more changes make them depart from the starting point:



Then, finally, one day, probably weeks or even months after they started, they decide to merge back:



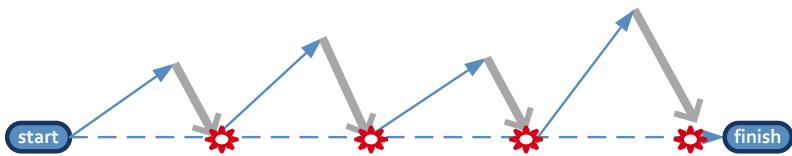
It wouldn't be very problematic if only you did this; but everybody else in the project was doing exactly the same and you end up with a huge integration problem. You have hit a Big Bang integration.



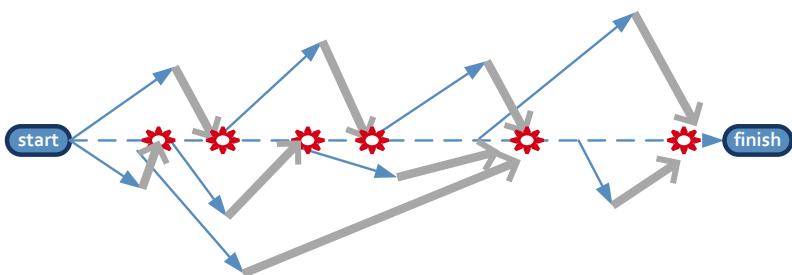
This is the original reason behind the move from waterfall to an incremental process, the key reason for agile, continuous integration, task branches and almost any other modern practice. Projects working this way end up spending far more time during the integration phase than during the actual development.

The solution, though, is theoretically simple: if there is a painful step in the process, do it as early as possible and as frequently as possible until it is no longer a problem.

Individuals share their changes back as soon as possible:



And, the entire team does the same on a frequent basis:



And, as a result, frequent releases are created as a continuous flow of incremental improvement.



Big Bang no more.

Enter task branches, continuous integration, iterative software constructions and incremental delivery.

APPENDIX D: 2 PRINCIPLES FOR PROJECT MANAGEMENT

There are two principles I consider basic for project management and are deeply rooted in the "task branch" way of thinking: have a fallback solution and provide great visibility.

Always have a fallback solution

I think I first read about it in "Rapid Development" by Steve McConnell. Anyway, since I first heard about this principle, it became my North Star for project management.

Things will go wrong, tasks will delay, everything will be harder than initially thought. And still, you'll have deadlines to meet and promises to keep.

The best solution to this problem is having a fallback solution, and the entire task branch strategy is built around that.

Suppose you create weekly releases. Nowadays weekly falls short and daily sounds much better, or even a few times a day. Anyway, let's stick to weekly releases.

If you are on a delivery week and this week's release is broken, instead of rushing out to do last minute changes, you will be better served by getting last week's stable release. You will be able to negotiate, explain what is not there, and protect your team from long hours and endless weekends.

Everybody prefers to have 100% of the 80% than 80% of nothing.



The first option shows two complete features and one not even started. You can deliver two features and negotiate. You have a fallback. The second option shows you almost have everything done, but nothing is complete. You can't deliver anything. You don't have a fallback. Don't be in that situation.

Maximize visibility

Ensure your customers, bosses, investors, colleagues, and anyone potentially interested in your project clearly sees that you are making progress.

Shout out loud and clear what you do, so everyone can see.

It might not sound humble but it is much better than the opposite: silence.

Silence must be good for unsung heroes in movies (well, they are in a movie anyway) but not that good for projects. Silence creates anxiety, alarm and agitation.

It is better that you report progress before you are asked to provide it.

Regular, understandable and precise progress is crucial, in my opinion, for any project.

And there's no better progress reporting than working software. Documents and PowerPoints come and go, but working software stays.

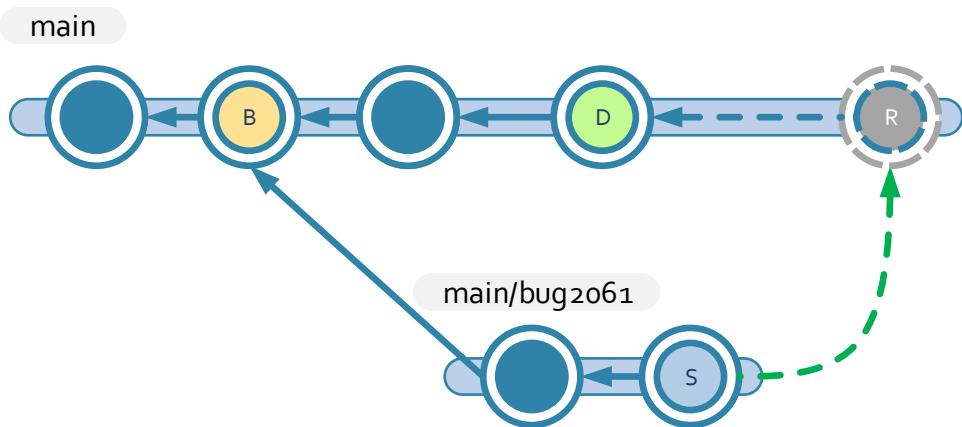
Here is how everything comes together. Task branches are key to creating frequent stable releases. Frequent stable releases create lots of undeniable visibility. And visibility builds trust.

Version Control, DevOps and Agile Development with Plastic SCM

Version control is the operating system of software development. It is the platform that supports all the other tools. This includes Continuous Delivery tools, testing frameworks, project management systems and even IDEs and editors. Used correctly, version control boosts team collaboration and productivity, and boosts team morale.

This book is ideal for developers, team leaders and development managers who need to learn not only how to adopt Plastic SCM but proven version control practices in general.

The book focuses not only on the specifics of Plastic, but also explains different branching patterns and ways of working, with a focus on how to combine task branches and trunk based development to implement DevOps successfully.



Pablo Santos Luaces founded Códice Software in 2005 and has been designing version control products for nearly 15 years. He plays several roles including core engineering, product design, marketing, business development, advertising and sales operations. He had helped implement Plastic SCM to hundreds of teams worldwide.

@psluaces
es.linkedin.com/in/psantosl

www.plasticscm.com
@plasticscm
facebook/plasticscm