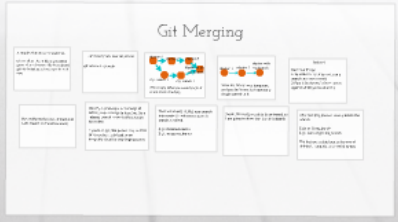
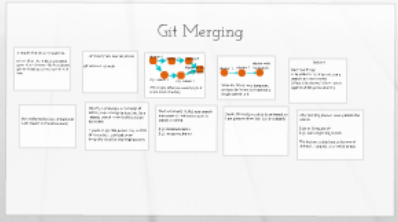
A Brief Guide to Version Control with Git



A Brief Guide to Version Control with Git

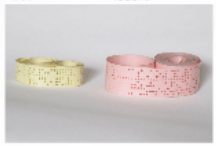


In the Beginning...

In the beginning was the code, and the code was with the program, and the program was the code.

In other words, you kept it on a disk. You updated it. After updating, maybe you stored in on another disk, or other storage medium.

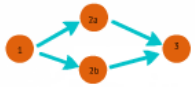
Version 1 Version 2



This was suboptimal.

Then in 1972, along came SCCS, the Source Code Control System! You could store multiple versions of a file on a hard drive. If you messed up, you could go back in time to when the code worked. You could see the evolution of a program. You might even store a different version than is currently running.

In 1982... RCS (Revision Control System) destroyed SCCS's dominance. You could have different branches, instead of a linear model. You could then merge these branches. This made working with multiple people much easier.



RCS and SCCS could only work on a single file at a time. This was fixed with the introduction of CVS (Concurrent Versioning System) which was originally just some scripts working with RCS to let you work with multiple files.

In the '80s, version control blossomed. There are now many version control systems available, all with their own benefits, drawbacks, and quirks.

BitKeeper, Perforce, Subversion, Visual SourceSafe, Mercurial, IBM ClearCase, AccuRev, AutoDesk Vault, Team Concert, Veritc, CVSNT, OpenCVS, Aegis, ArX, Darcs, Fossil, GNU Arch, BitKeeper, Code Co-Op, Plastic, StarTeam, MKS Integrity, Team Foundation Server, PVCS, DCVS, StarTeam, Vesscity, Razor, Sun TeamWare, Code Co-Op, SVK, Fossil, Codeville, Bazaar...

... but we're going to talk about...

Git

In the beginning was the code, and the code was with the program, and the program was the code.

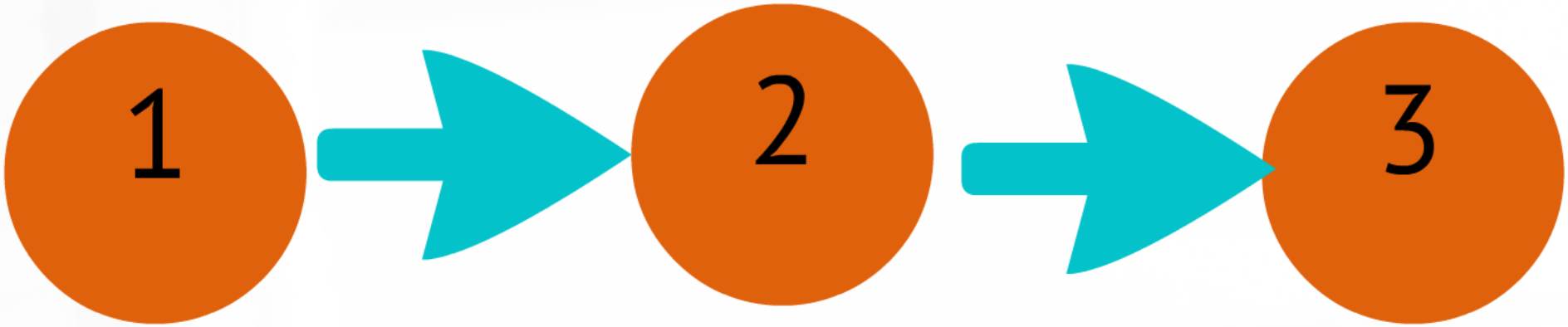
In other words, you kept it on a disk. You updated it. After updating, maybe you stored in on another disk, or other storage medium.

Version 1

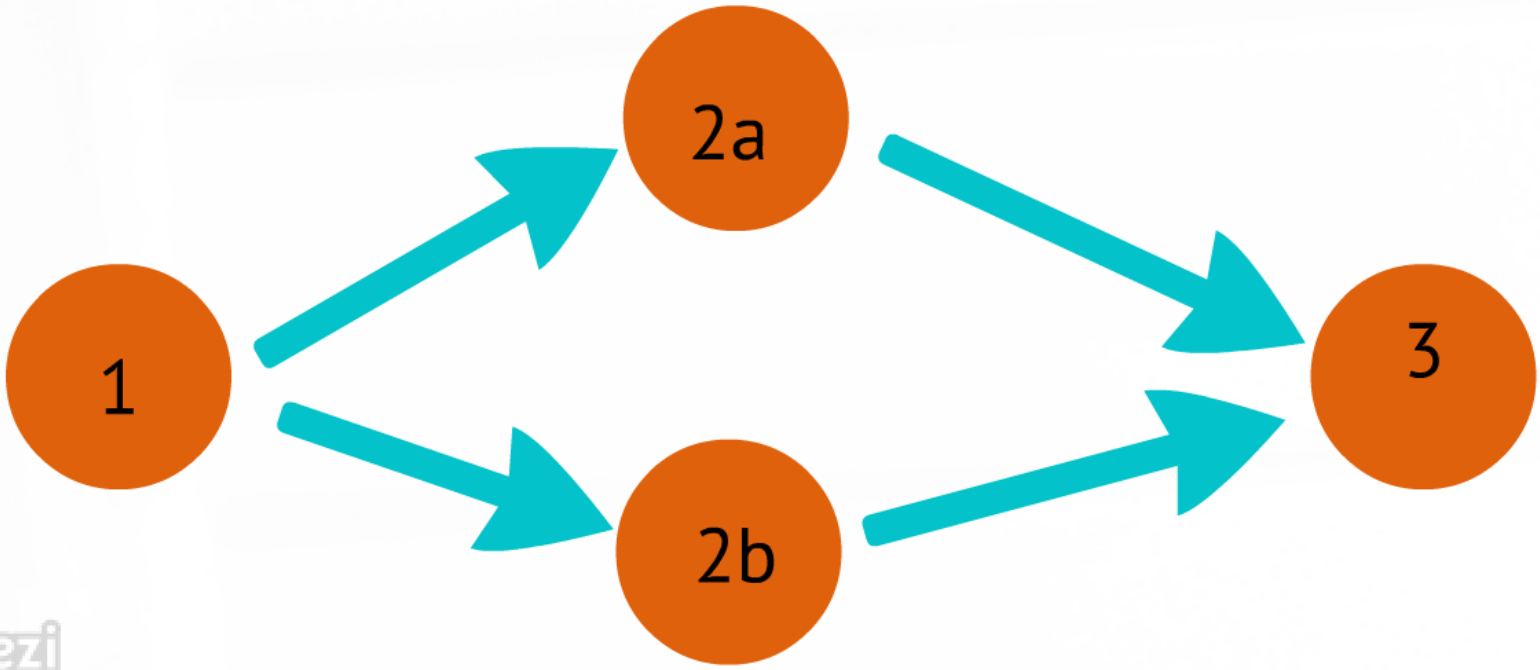
Version 2

This was suboptimal.

Then in 1972, along came SCCS, the Source Code Control System! You could store multiple versions of a file on a hard drive. If you messed up, you could go back in time to when the code worked. You could see the evolution of a program. You might even store a different version than is currently running.



In 1982... RCS (Revision Control System) destroyed SCCS's dominance. You could have different branches, instead of a linear model. You could then merge these branches. This made working with multiple people much easier.



RCS and SCCS could only work on a single file at a time. This was fixed with the introduction of CVS (Concurrent Versioning System) which was originally just some scripts working with RCS to let you work with multiple files.

In the '80s, version control blossomed. There are now many version control systems available, all with their own benefits, drawbacks, and quirks.

BitKeeper, Perforce, Subversion, Visual SourceSafe, Mercurial, IBM ClearCase, AccuRev, AutoDesk Vault, Team Concert, Vesta, CVSNT, OpenCVS, Aegis, ArX, Darcs, Fossil, GNU Arch, BitKeeper, Code Co-Op, Plastic, StarTeam, MKS Integrity, Team Foundation Server, PVCS, DCVS, StarTeam, Veracity, Razor, Sun TeamWare, Code Co-Op, SVK, Fossil, Codeville, Bazaar....

... but we're going to talk about...

Git

Git Basics

Git is great.

Really, really great.

But a bit weird.

Git

Developed by Linus Torvalds
Strong support for distributed development
Very fast
Very efficient
Very resistant against data corruption
Makes branching and merging easy
Can run over various protocols
Native version control for Github

Git is a distributed version control system.

First, some terminology.

repository (or repo) - where your code is stored.
branch - the current set of code you are working with. A repo usually has multiple branches.

Example

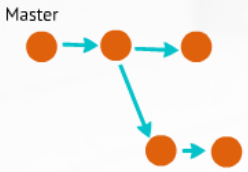
Rent-A-Cat has a:
MainWebsite repo. Code for the Rent-A-Cat website.
DataWarehouse repo. Code for their data warehouse.
MobileApp repo. Code for the mobile Cat Rental app.

Example

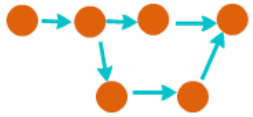
There is a master branch which contains a version of the repo which is running on the main site. You don't want to edit this!

Example

You are adding a feature to display cat images, instead of just describing them in text. You can make a display_cat_images branch which is a copy of master, but changes you make won't be seen on master.



You can merge this code in to the master branch when you're done.



Well, conceptually, yes, but not exactly.

These circles are snapshots of code in time. They are called commits.
The arrows are differences between commits. We call them deltas.

So when you're working, you "make a commit" - that it, make a snapshot of what you're currently working on.

$ git commit -a

Git stores this internally as the change from the last commit.

Commits are identified by SHAs - a large hexadecimal number created by the SHA1 hash function.

Git is also distributed, which means that the code you are working on will be different than what others are working on.
Almost always work on different branches, so you don't overlap with people.

This means that you have a complete copy on your local machine. There is usually one machine designated as the "true" copy, often called "origin".

You can PUSH your changes to origin, or PULL the changes from origin.

If you want to modify someone else's code, you can issue a PULL REQUEST (say, "hey, pull my code into your repo").

Example

```
$ git checkout master
$ git pull
$ git checkout -b my_branch
$ emacs foo.rb
$ git commit -a
$ git push origin my_branch
```

Github is just a really big git server. You can have your own git server at home and call it whatever you want.

But let's do a quick example with Github.

Example

Git is great.

Really, really great.

But a bit weird.

Git

Developed by Linus Torvalds

Strong support for distributed development

Very fast

Very efficient

Very resistant against data corruption

Makes branching and merging easy

Can run over various protocols

Native version control for Github

Git is a distributed version control system.

First, some terminology.

repository (or repo) - where your code is stored.
branch - the current set of code you are working with. A repo usually has multiple branches.

Example

Rent-A-Cat has a:

MainWebsite repo. Code for the Rent-A-Cat website.

DataWarehouse repo. Code for their data warehouse.

MobileApp repo. Code for the mobile Cat Rental app.

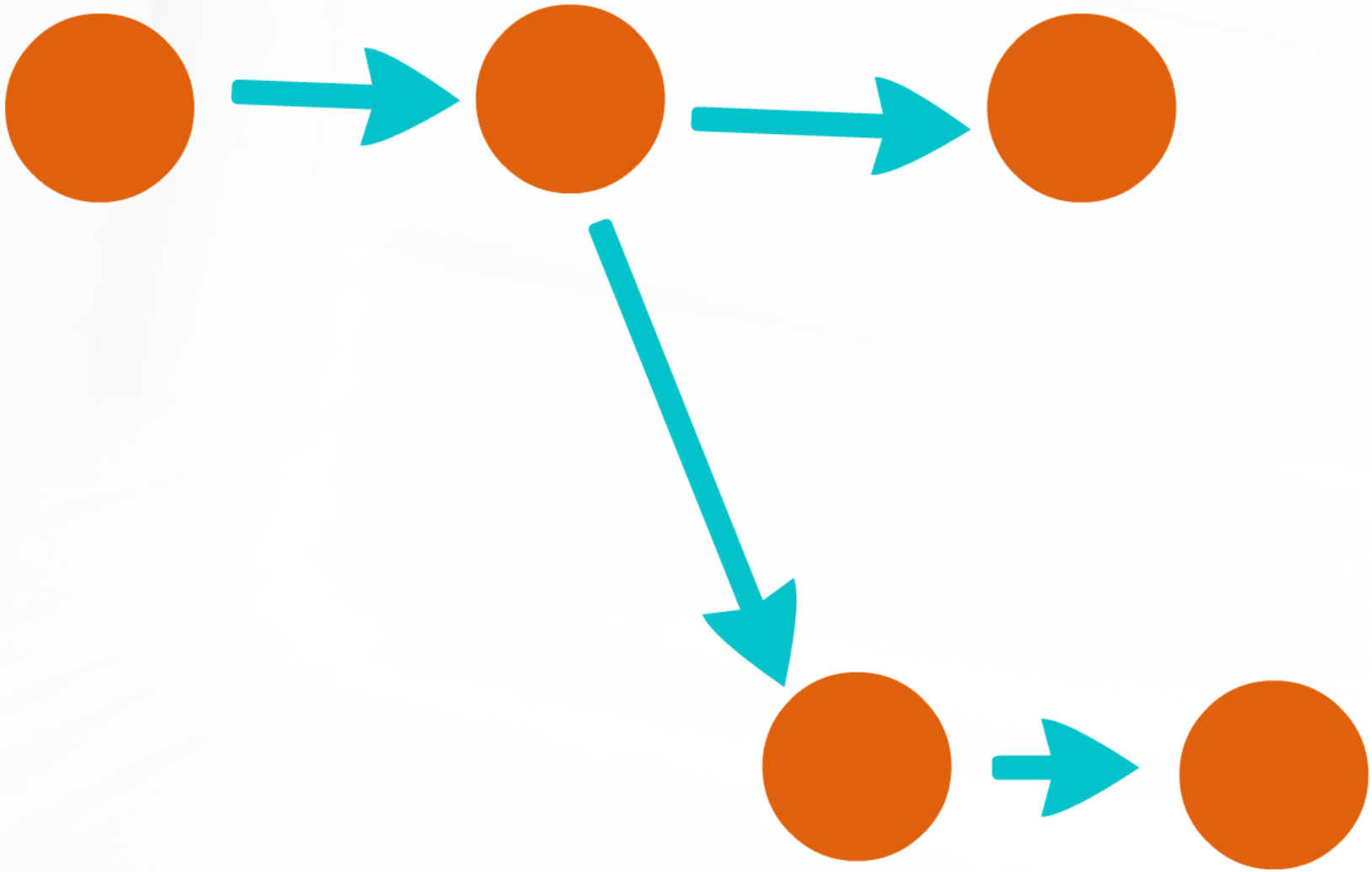
Example

There is a master branch which contains a version of the repo which is running on the main site. You don't want to edit this!

Example

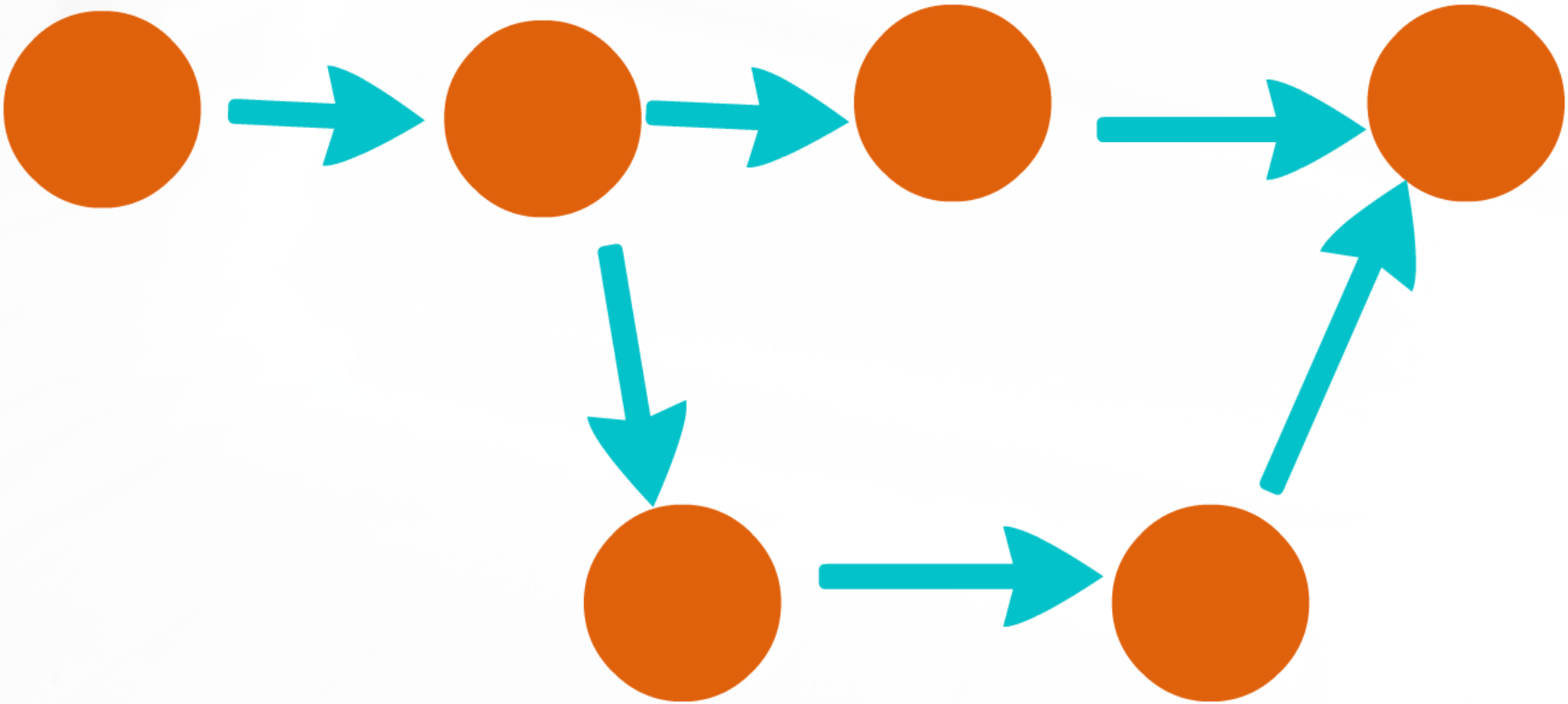
You are adding a feature to display cat images, instead of just describing them in text. You can make a display_cat_images branch which is a copy of master, but changes you make won't be seen on master.

Master

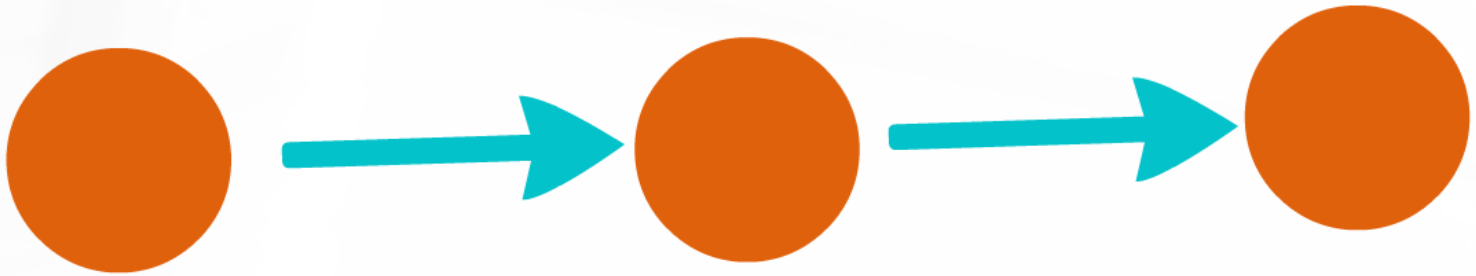


display_cat_images

You can merge this code in to the master branch when you're done.



Well, conceptually, yes, but not exactly.



These circles are snapshots of code in time. They are called commits.
The arrows are differences between commits. We call them deltas.

So when you're working, you "make a commit" - that it, make a snapshot of what you're currently working on.

$ git commit -a

Git stores this internally as the change from the last commit.

Commits are identified by SHAs - a large hexadecimal number created by the SHA1 hash function.

Git is also distributed, which means that the code you are working on will be different than what others are working on.

Almost always work on different branches, so you don't overlap with people.

This means that you have a complete copy on your local machine. There is usually one machine designated as the "true" copy, often called "origin".

You can PUSH your changes to origin, or PULL the changes from origin.

If you want to modify someone else's code, you can issue a PULL REQUEST (say, "hey, pull my code into your repo").

Example

```
$ git checkout master
$ git pull
$ git checkout -b my_branch
$ emacs foo.rb
$ git commit -a
$ git push origin my_branch
```

Github is just a really big git server. You can have your own git server at home and call it whatever you want.

But let's do a quick example with Github.

Example

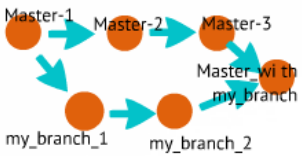
Git Merging

A couple of other command first..

git add <file> - Add a file to git control
git rm <file>- Remove a file from git ctrl
git init - Initialize a directory to be a git repo

.. and everyone's favorite, rebase

git rebase -i <branch>



This is ugly. What you would prefer is a nice linear structure.

Since my_branch was temporary, and you don't need to know every single commit in it.

Rebase!

Does two things:
1. Squashes all your commits on a branch into one commit
2. Puts it "at the end" of the branch against which you're rebasing

(this really needs to be animated, so I will draw it on the whiteboard)

Usually, a gatekeeper is in charge of letting people merge to baseline. So a rebased branch is sent to that person for review.

If you're in QA, that person may be YOU. Or it may be a tech lead, or an integration lead for very large projects.

That person will MERGE your branch into master (or whatever the main branch is called).

$ git checkout master
$ git merge my_branch

(again, this really needs to be animated, so I am going to draw it on the whiteboard)

After merging, you can usually delete the branch.

$ git br -D my_branch
$ git push origin :my_branch

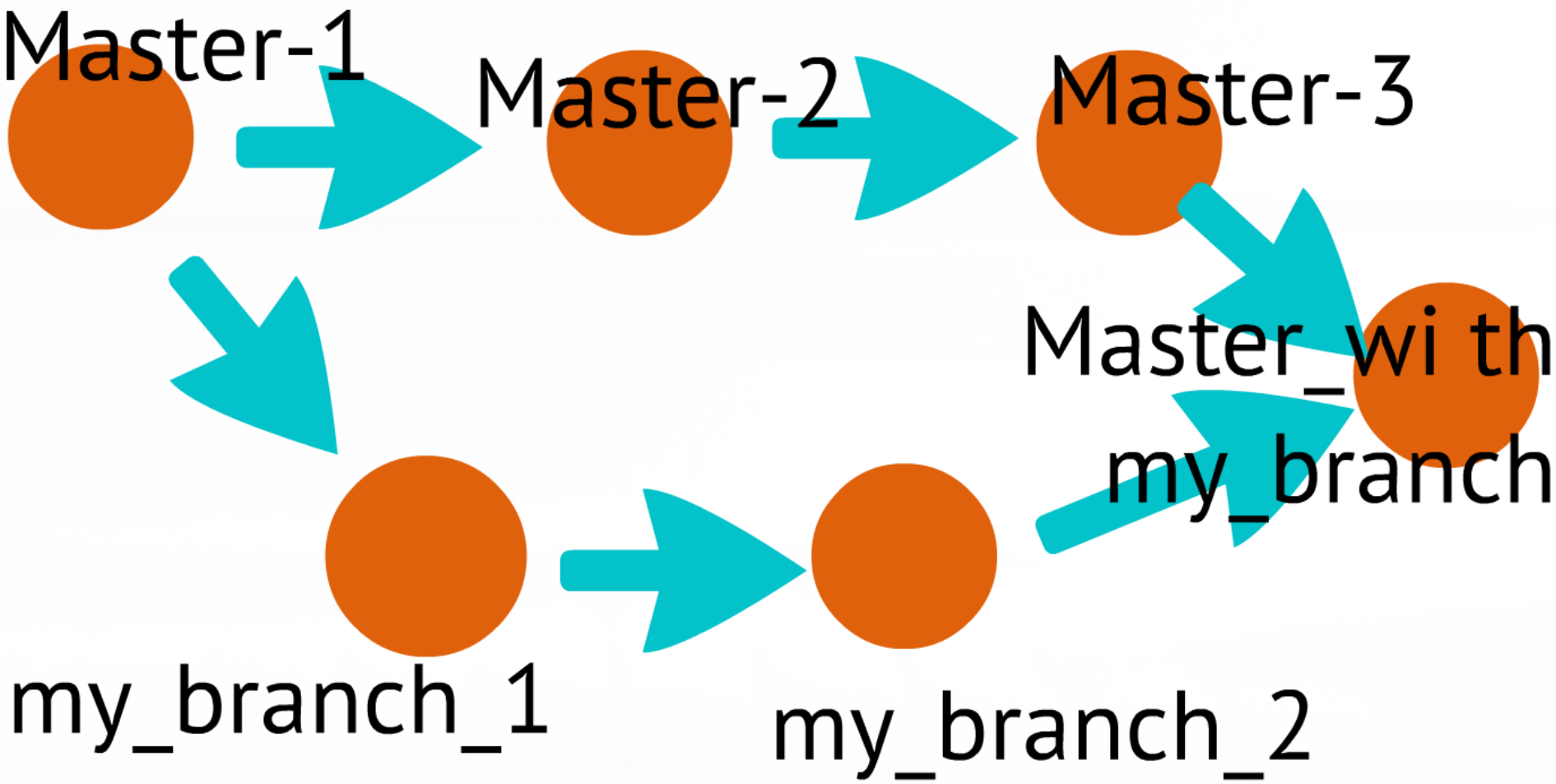
The first one delete locally, the second deletes it remotely. It's a weird syntax.

A couple of other command first..

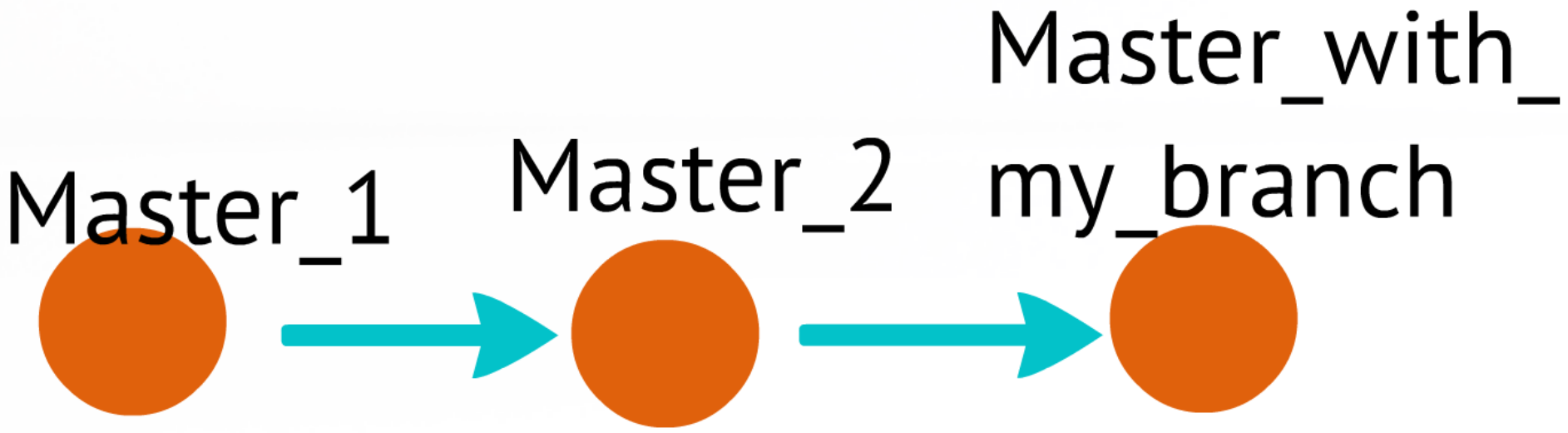
git add <file> - Add a file to git control
git rm <file>- Remove a file from git ctrl
git init - Initialize a directory to be a git repo

.. and everyone's favorite, rebase

git rebase -i <branch>



This is ugly. What you would prefer is a nice linear structure.



Since my_branch was temporary, and you don't need to know every single commit in it.

Rebase!

Does two things:

1. Squashes all your commits on a branch into one commit
2. Puts it "at the end" of the branch against which you're rebasing

(this really needs to be animated, so I will draw it on the whiteboard)

Usually, a gatekeeper is in charge of letting people merge to baseline. So a rebased branch is sent to that person for review.

If you're in QA, that person may be YOU. Or it may be a tech lead, or an integration lead for very large projects.

That person will MERGE your branch into master (or whatever the main branch is called).

```
$ git checkout master
$ git merge my_branch
```

(again, this really needs to be animated, so I am going to draw it on the whiteboard)

After merging, you can usually delete the branch.

```
$ git br -D my_branch
$ git push origin :my_branch
```

The first one delete locally, the second deletes it remotely. It's a weird syntax.

Git Tips and Tricks

git status - see what has happened, what branch you're in

curb - See current branch only

```
alias curb="git branch | grep '*' | sed s/*// | sed s/\ //"
```

git diff

See what has changed since last commit

Find a GUI

(gitg, tig, others)

Good for seeing complex interactions

Never, ever re-write public history if other people depend on it.

Never, ever. EVER.

In other words, if you push a branch to origin, don't re-push to it, unless you know no one else is using it. Make a new one.

If you need to do so..

git push --force

Commit early, commit often.

Branching is cheap.

Do it often.

Come up with a branch naming convention, e.g.

initials_feature_description

wjl_bill_awesomeness_refactor

or feature number _ branch descriptor
US007_bill_FOR_REVIEW

git status - see what has happened, what branch you're in

curb - See current branch only

```
alias curb="git branch | grep '*' | sed s/*// | sed s/^ //"
```

git diff

See what has changed since last commit

Find a GUI

(gitg, tig, others)

Good for seeing complex interactions

Never, ever re-write public history if other people depend on it.

Never, ever. EVER.

In other words, if you push a branch to origin, don't re-push to it, unless you know no one else is using it. Make a new one.

If you need to do so..

git push --force

Commit early,
commit often.

Branching is cheap.

Do it often.

Come up with a branch naming convention, e.g.

initials_feature_description

wjl_bill_awesomeness_refactor

or feature number _ branch descriptor
US007_bill_FOR_REVIEW

Any questions?

A Brief Guide to Version Control with Git