Git

"all meaningful operations can be expressed in terms of the `rebase` command"
--
Linus Torvalds, 2015

a talk by alum Ross Schlaikjer
for the GNU/Linux Users Group

Sound familiar?

add
commit

diff

log

init
clone

push
pull

status

branch
merge

Git only know a handful of tricks

Once you know them, it’s quite simple

Command names don’t give away what they do

But it all sorta make sense if you know internals

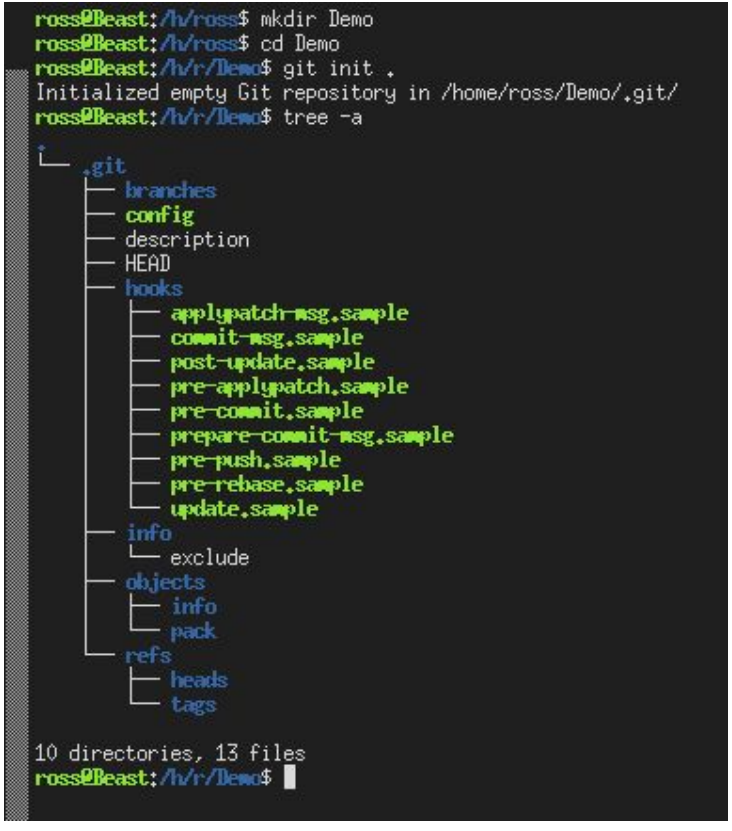
(and the internals are good. Linus is just *not* an interface designer)

Getting Started

git init

git clone

Lets make a repository



Getting stuff done

git add

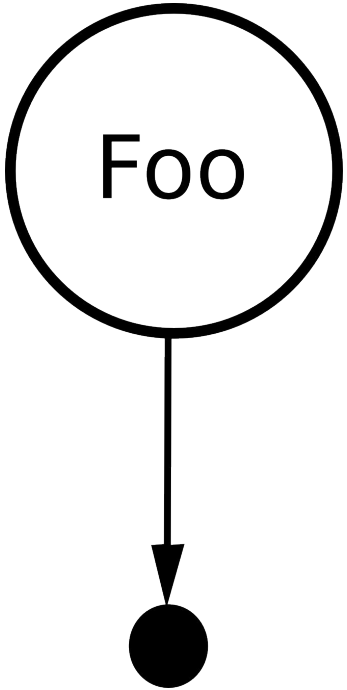
git commit

Getting stuff done

```
ross@Beast:/h/r/Demo$ echo 'Hello git!' | tee foo
Hello git!
ross@Beast:/h/r/Demo$ git add foo
```

‘`git add`’ writes to the repository.

Creates a ‘binary object’



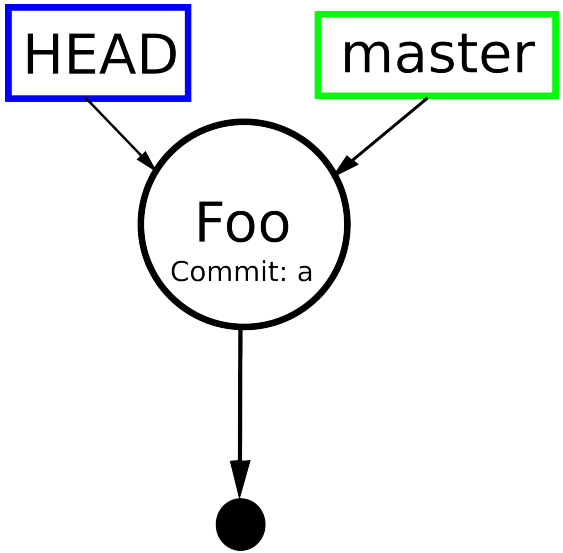
Getting stuff done

```
ross@Beast:/h/r/Demo$ git commit -m “Commitment”

Add wrote to the repo, commit does not
```

Commit:

- Creates a commit object with an ID
- Adds labels to it



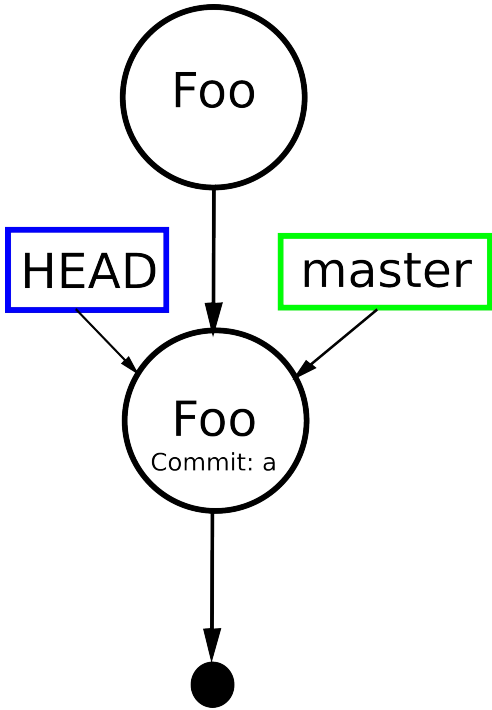
You’ve just seen 80% of git

Let’s add again

```
ross@Beast:/h/r/Demo$ echo 'Hello once more' | tee -a foo
Hello once more
ross@Beast:/h/r/Demo$ git add foo
```

Adds a new copy of foo to the repo

Doesn’t move the labels



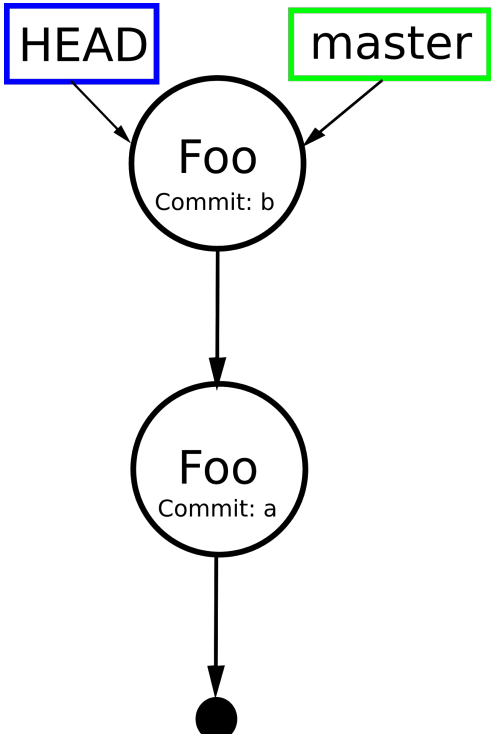
And commit

```
ross@Beast:/h/r/Demo$ git commit -m ‘Second commit!’
```

Creates another commit object

Move the labels

That’s it!



Branching

Branching is super easy!

What happens if we do:

```
ross@Beast:/h/r/Demo$ git branch feature
```

?

Branching

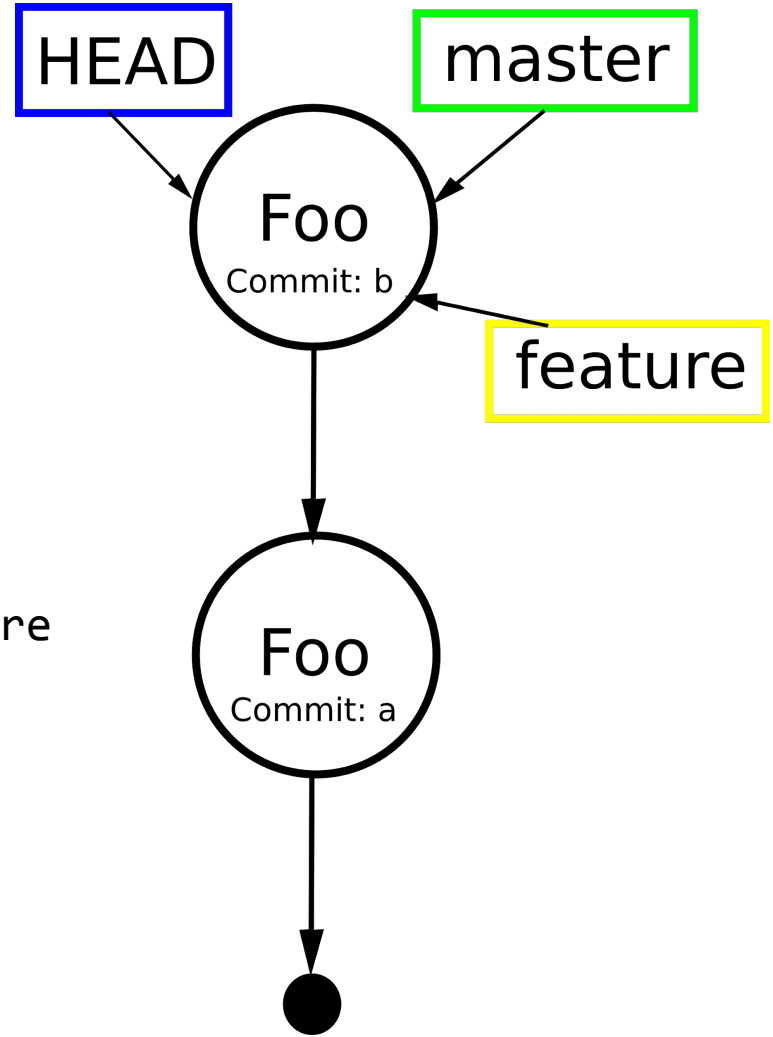
Branching is super easy!

What happens if we do:

```
ross@Beast:/h/r/Demo$ git branch feature
```

We add another label

That’s it!

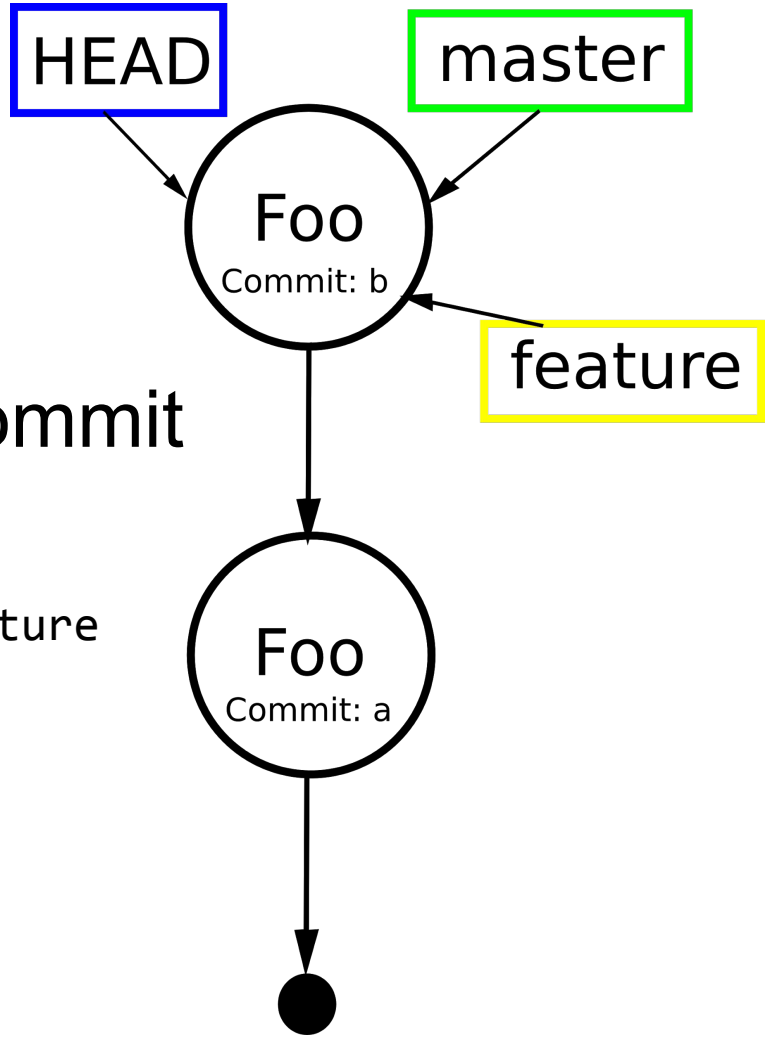


Checkout

Checking out code changes

which label moves when I commit

```
ross@Beast:/h/r/Demo$ git checkout feature
```



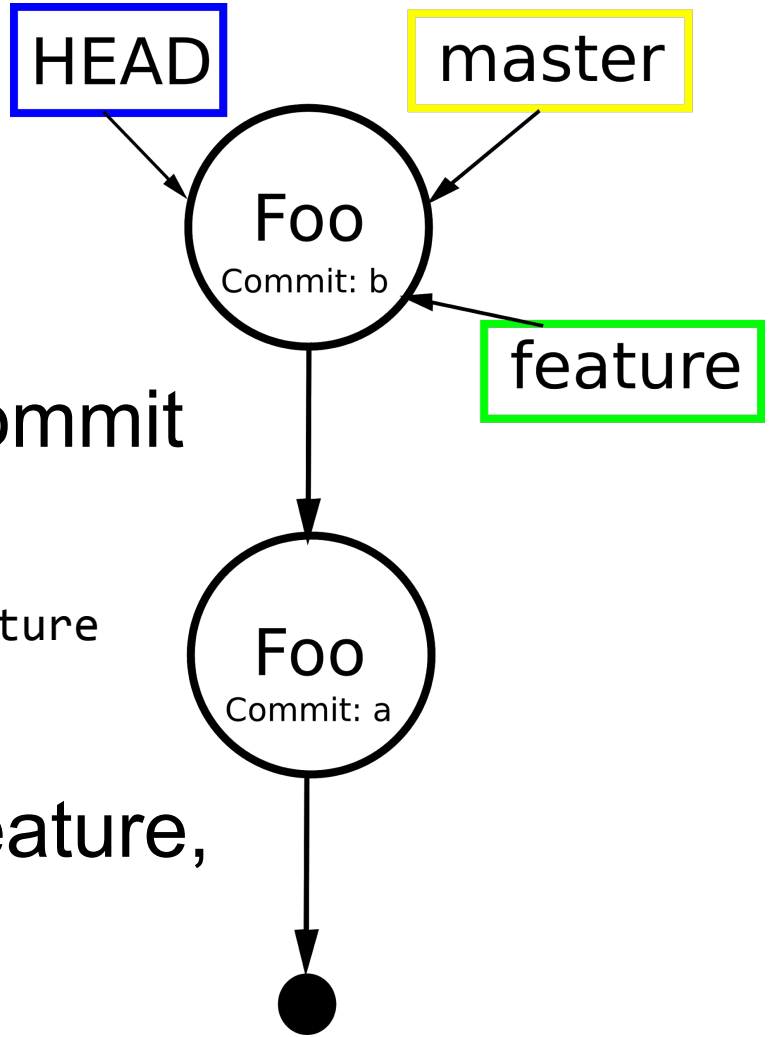
Checkout

Checking out code changes

which label moves when I commit

```
ross@Beast:/h/r/Demo$ git checkout feature
```

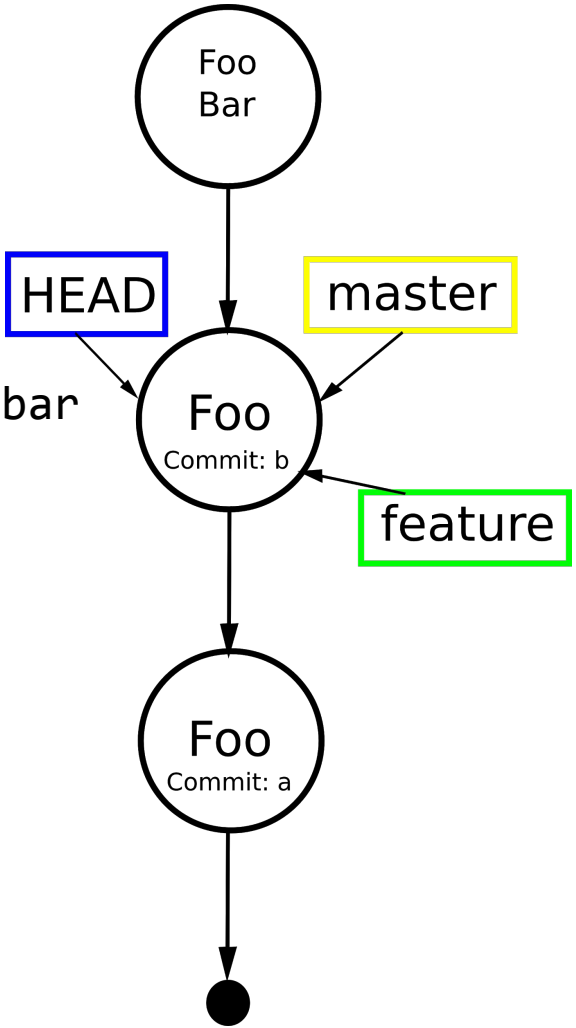
Committing will now move feature, not master



Branching

So let's add a new file

```
ross@Beast:/h/r/Demo$ echo "New branch" | tee bar
ross@Beast:/h/r/Demo$ git add bar
```



Branching

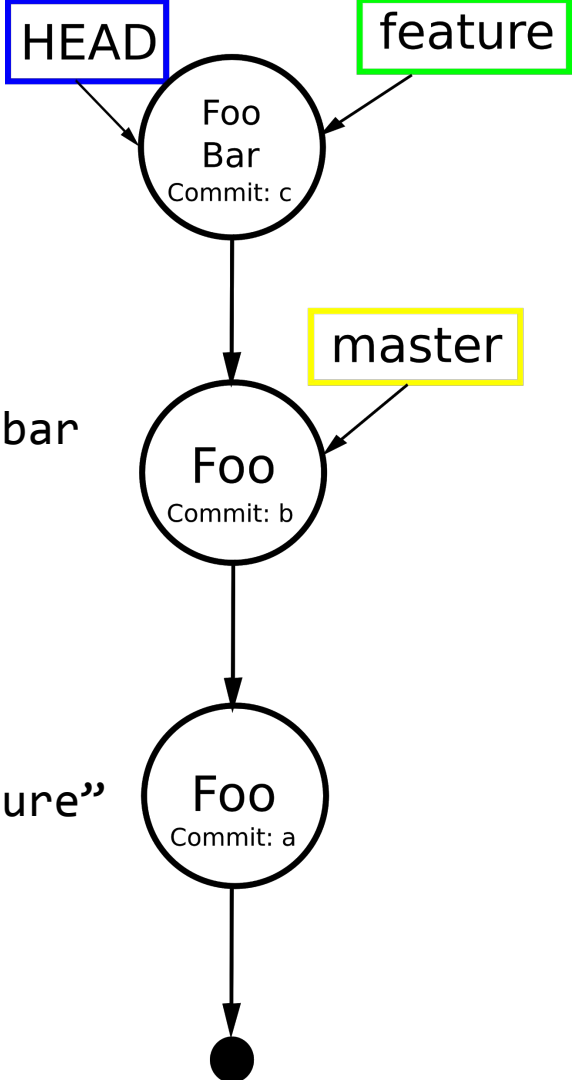
So let's add a new file

```
ross@Beast:/h/r/Demo$ echo "New branch" | tee bar
ross@Beast:/h/r/Demo$ git add bar
```

And commit

```
ross@Beast:/h/r/Demo$ git commit -m "cool feature"
```

That's branching!



Staging Area

aka ‘Index’
aka ‘Cache’

Staging area

When you add, git writes to a staging area

Commit commits the data in the staging area

Allows you to build up commits

Let’s talk IDs

Git IDs are SHA1sums

- What's a checksum
- Git SHAs contain:

  content (files), author, date, log message, previous commit
- Every ID is unique
- Every commit is unique
- Commits never change

Typical git workflow

branch feature

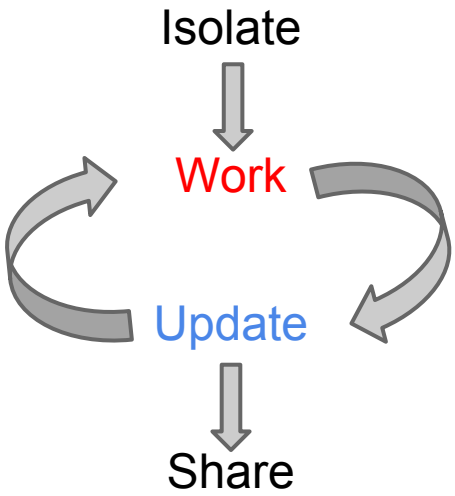
checkout feature

test

checkout master

merge feature

Typical git workflow (simpler)

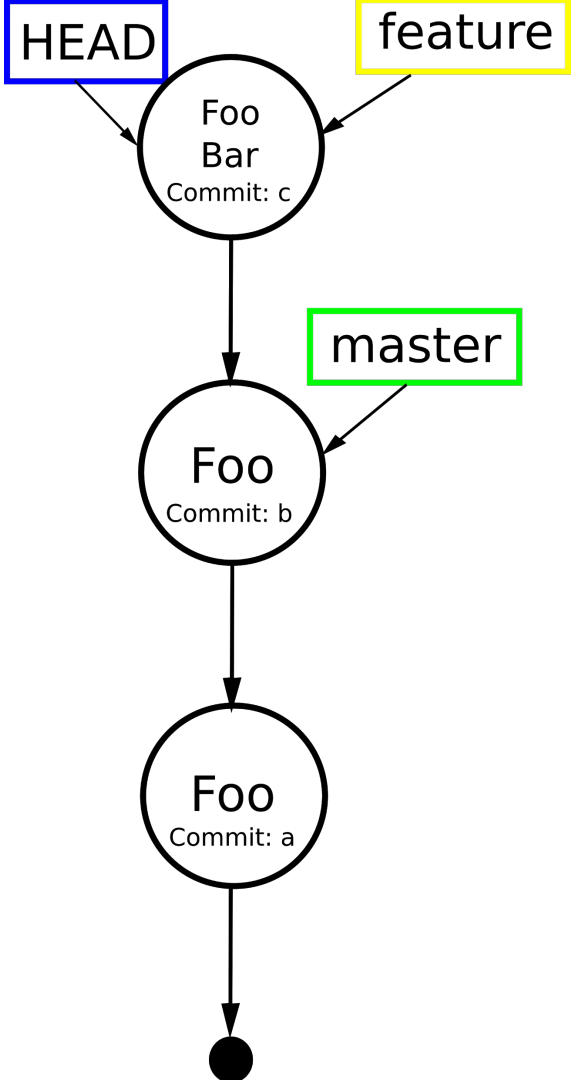


Merge

Feature is complete!

```
Demo$ git checkout master
Demo$ git merge feature
Updating 7cef3ce..e48af8d
Fast-forward
 bar | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 bar
```

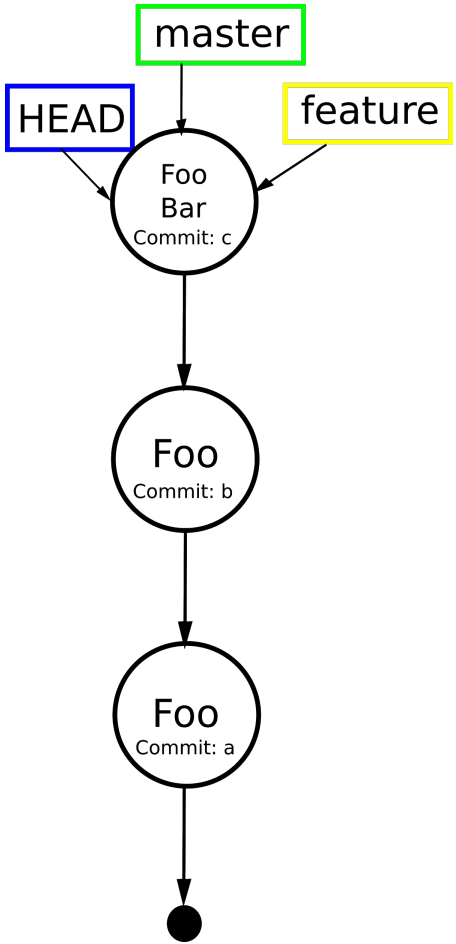
Special case: fast forward



Merge (fast-forward)

```
Demo$ git merge feature
Updating 7cef3ce..e48af8d
Fast-forward
 bar | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 bar
```

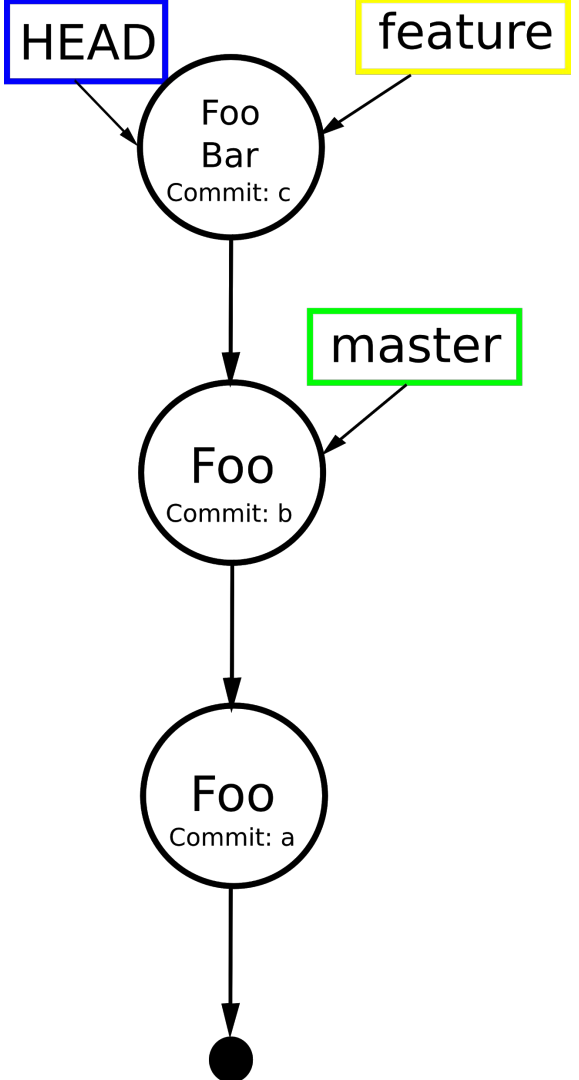
It’s all in a line, just move the label!



What if it’s not simple?

What if we change master?

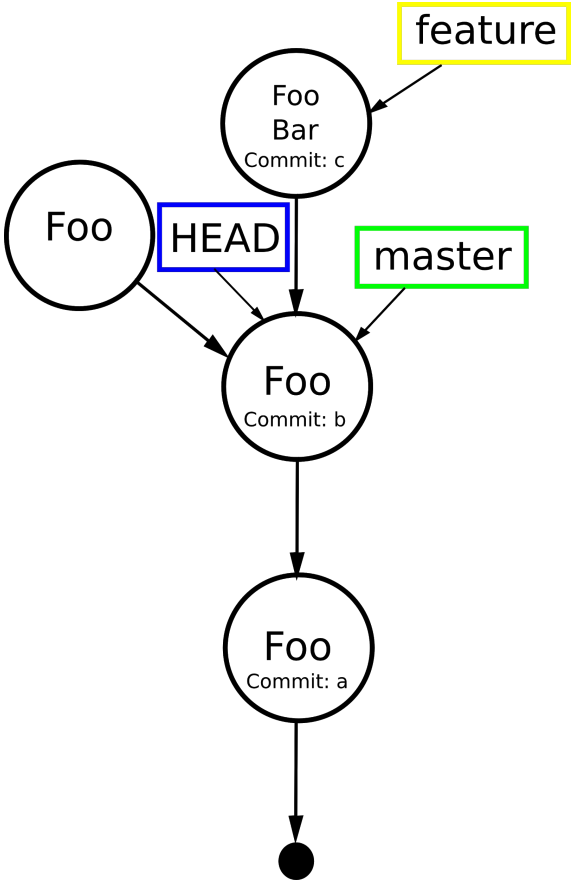
```
Demo$ git checkout master
Demo$ echo “Time for something new” > foo
Demo$ git add foo
```



What if it’s not simple?

What if we change master?

```
Demo$ git checkout master
Demo$ echo “Time for something new” > foo
Demo$ git add foo
```



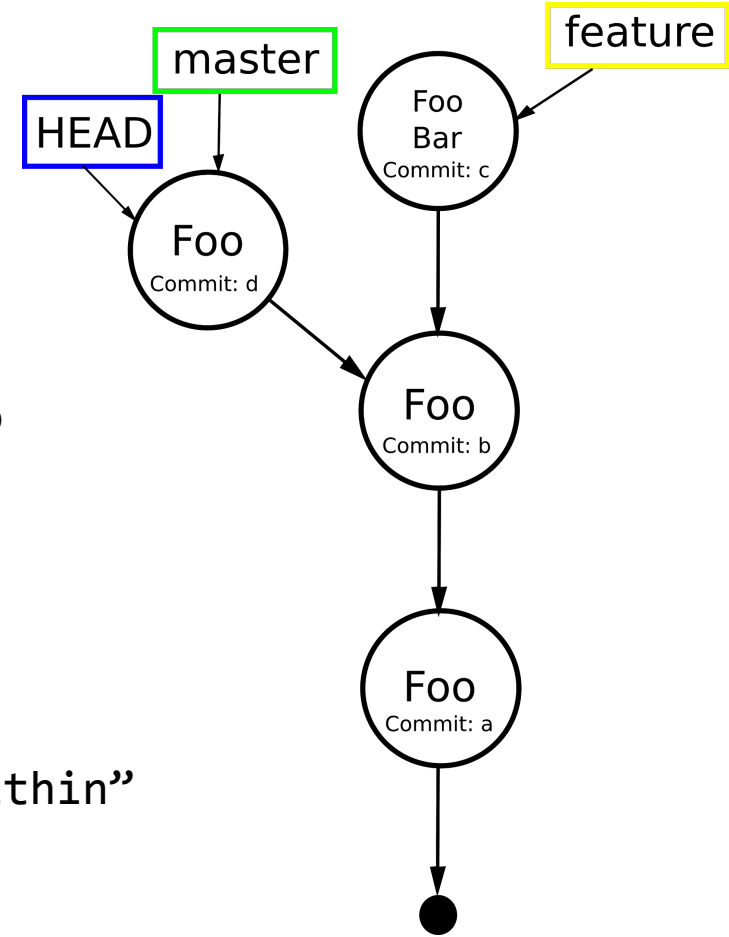
What if it’s not simple?

What if we change master?

```
Demo$ git checkout master
Demo$ echo “Time for something new” > foo
Demo$ git add foo
```

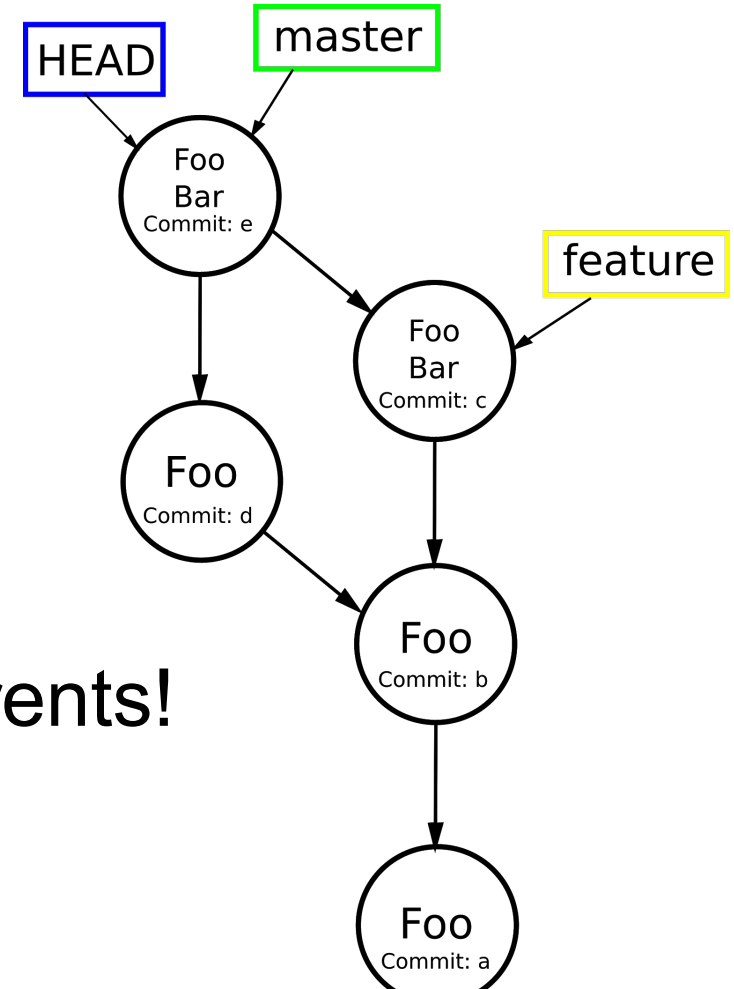
And commit:

```
Demo$ git commit -m “Change comes from within”
```



So now let's merge

```
Demo$ git merge feature
--asks for a commit message--
Merge made by the 'recursive' strategy.
 bar | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 bar
```



Merge commits have two parents!

Disaster strikes

Turns out I didn’t want to do that

Reset

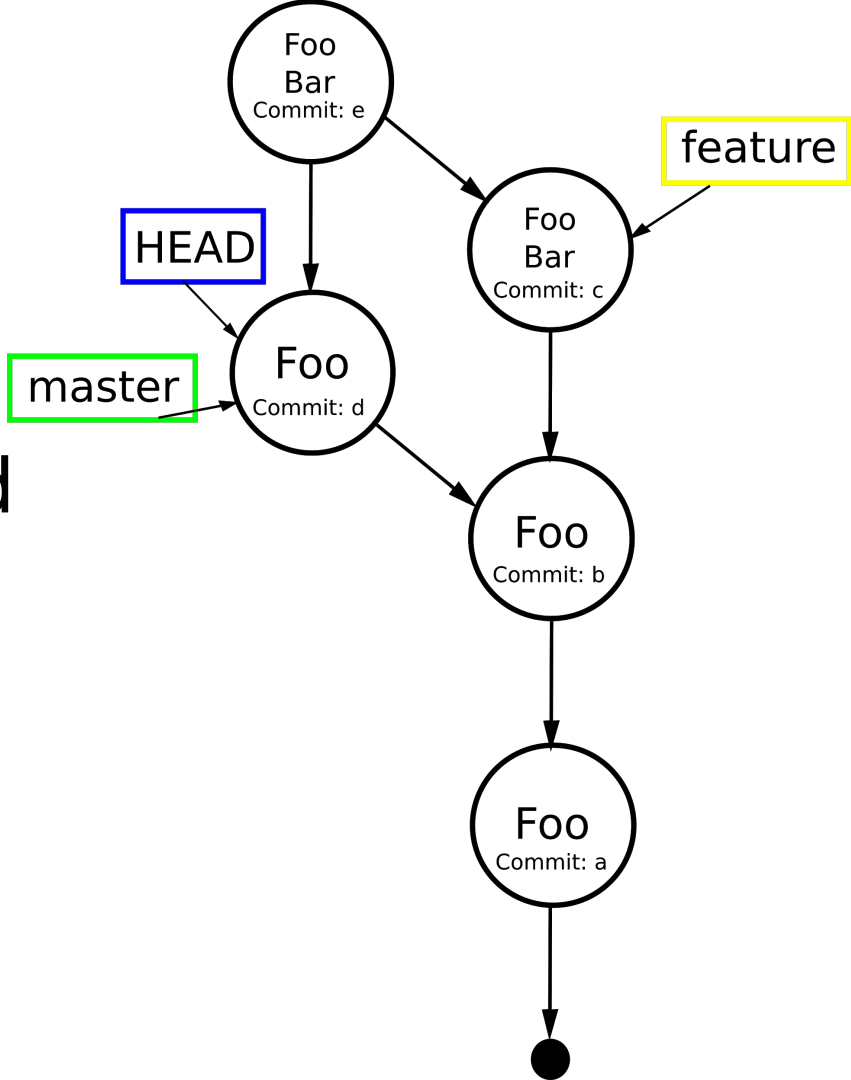
```
Demo$ git reset --hard HEAD^
```

Reset moves labels around

(--hard means checkout after moving)

Commit ID e still exists -

you can go back



Playing well with others

push

fetch

pull

Playing well with others

push

fetch

pull

This is where the trouble starts

Remotes

For sharing, you need to know where

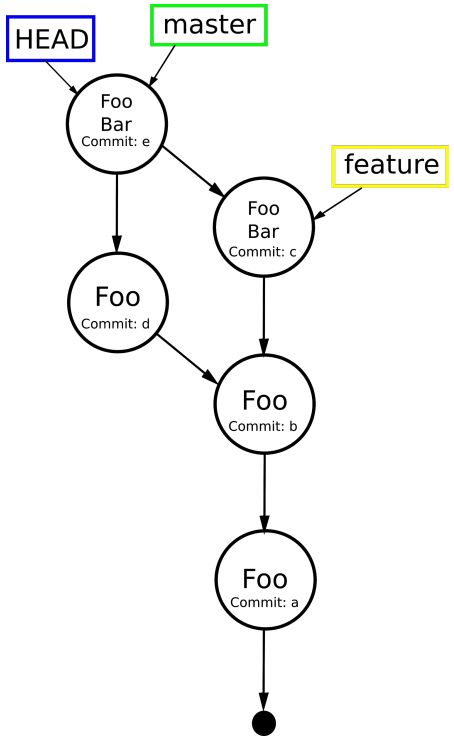
```
Demo$ mkdir ../Remote
Demo$ git init --bare ../Remote
Demo$ git remote add origin ../Remote
```

Git doesn’t care how it gets at the remote

(http, SSH, filesystem, git protocol)

So let’s share

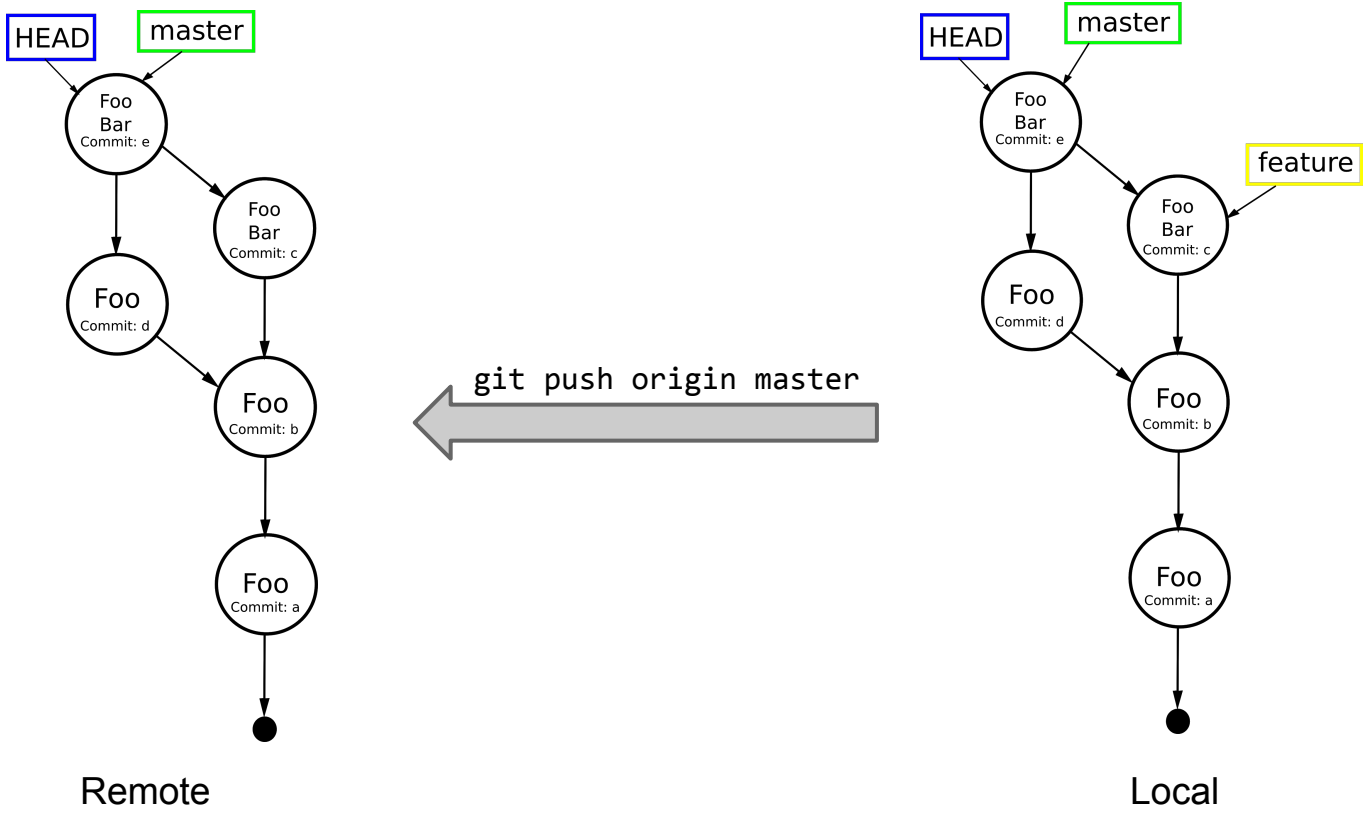
```
Demo$ git push origin master
```



Remote

Local

So let’s share



git push

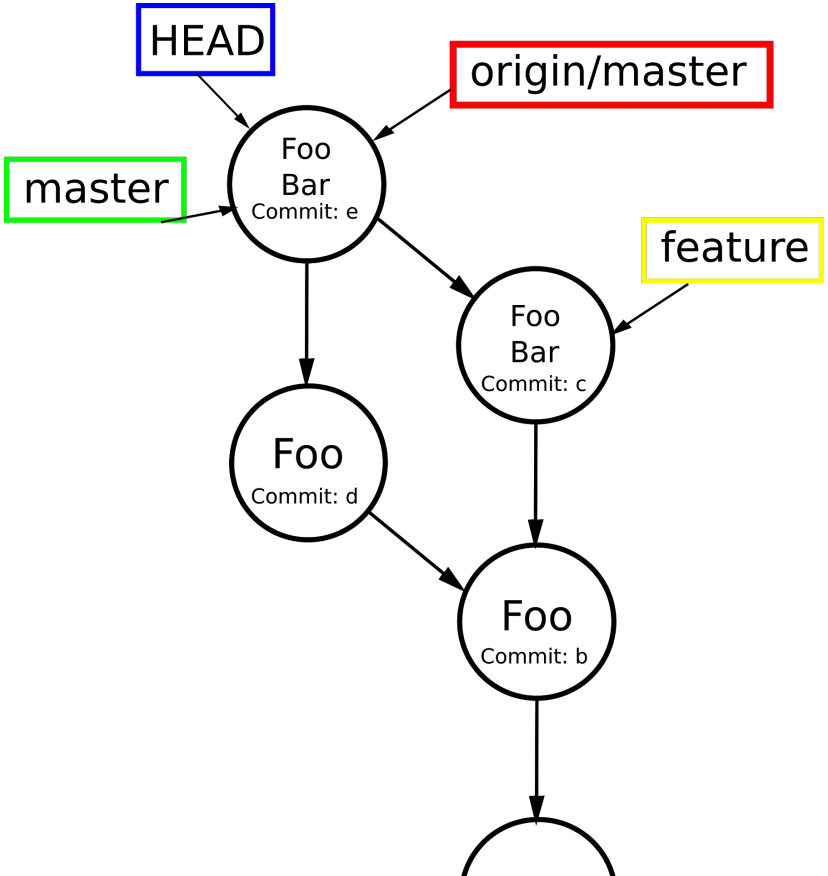
Like most git ops, works on current branch

Can do funky things -

```
// Push local branch feature to remote branch master
Demo$ git push origin feature:master
// Push nothing to remote branch feature (aka delete feature)
Demo$ git push origin :feature
```

Wait, what's that?

When you talk to master, git takes note of where it thinks things are



remote branch vs. remote/branch

remote/branch = local label (ref)

remote branch = the branch, on remote

diff, log, etc. want refs

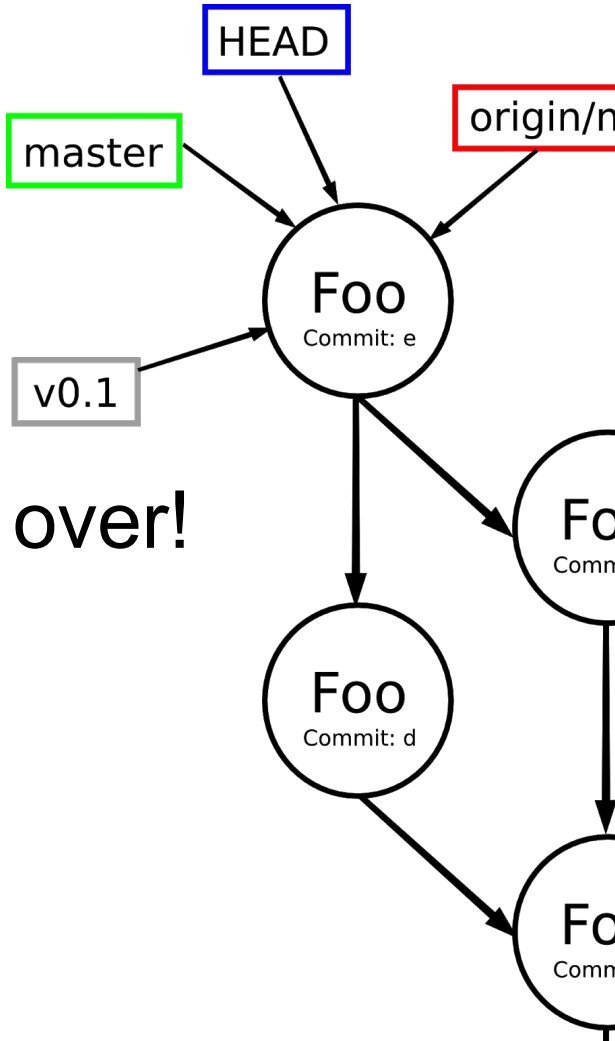
push wants to to know the remote and branch

tags

Label, just like everything else.

Git is just the same trick over and over!

Tags cannot move, however



Going along as normal

```
Demo$ git checkout master
Demo$ echo “Totally bug fee line of code” >> foo
Demo$ git commit -am “Everything is fine”
```

Everything is not fine

```
Demo$ git checkout master
Demo$ echo “Totally bug fee line of code” >> foo
Demo$ git commit -am “Everything is fine”
```

I made an error in code I already committed!

I guess I’ll just fix it and commit again

```
Demo$ sed -i.bak ‘s/fee/free/g’ foo
Demo$ git commit -am “Typo fix”
Demo$ git push
```

Rebase

Rebase

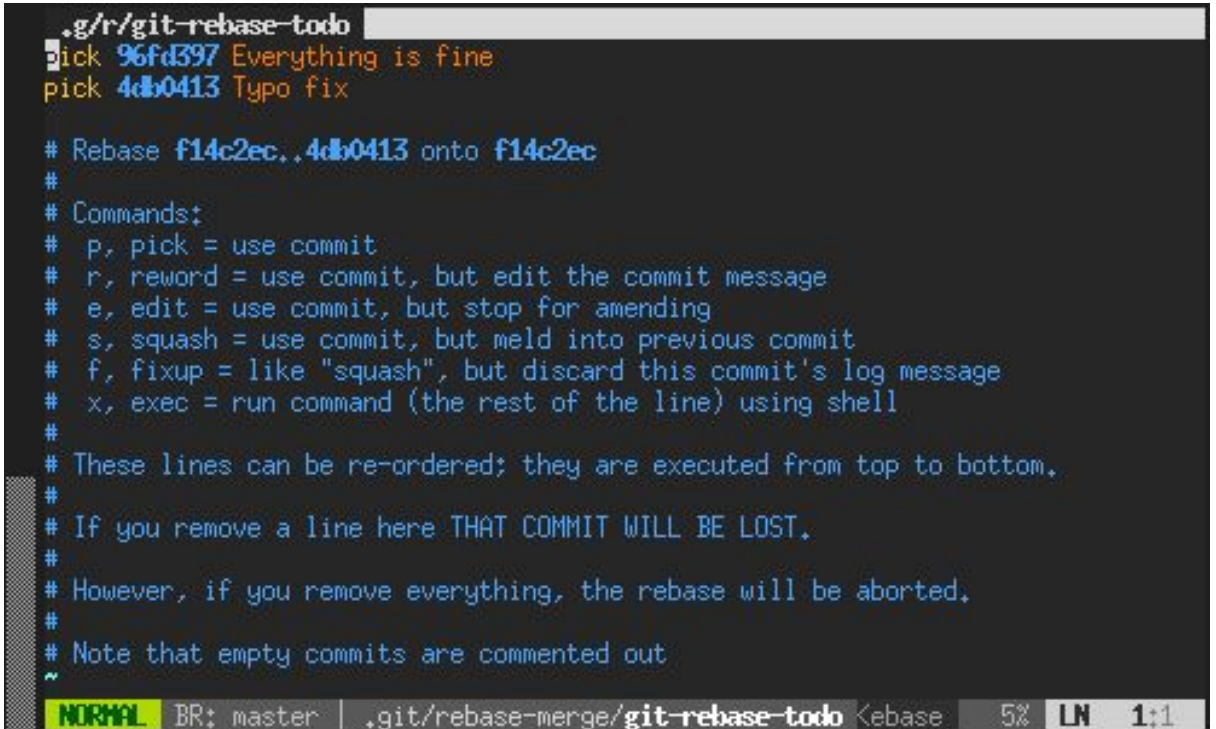
Often described as ‘rewrites history’

But history is immutable!

It really creates a whole new history.

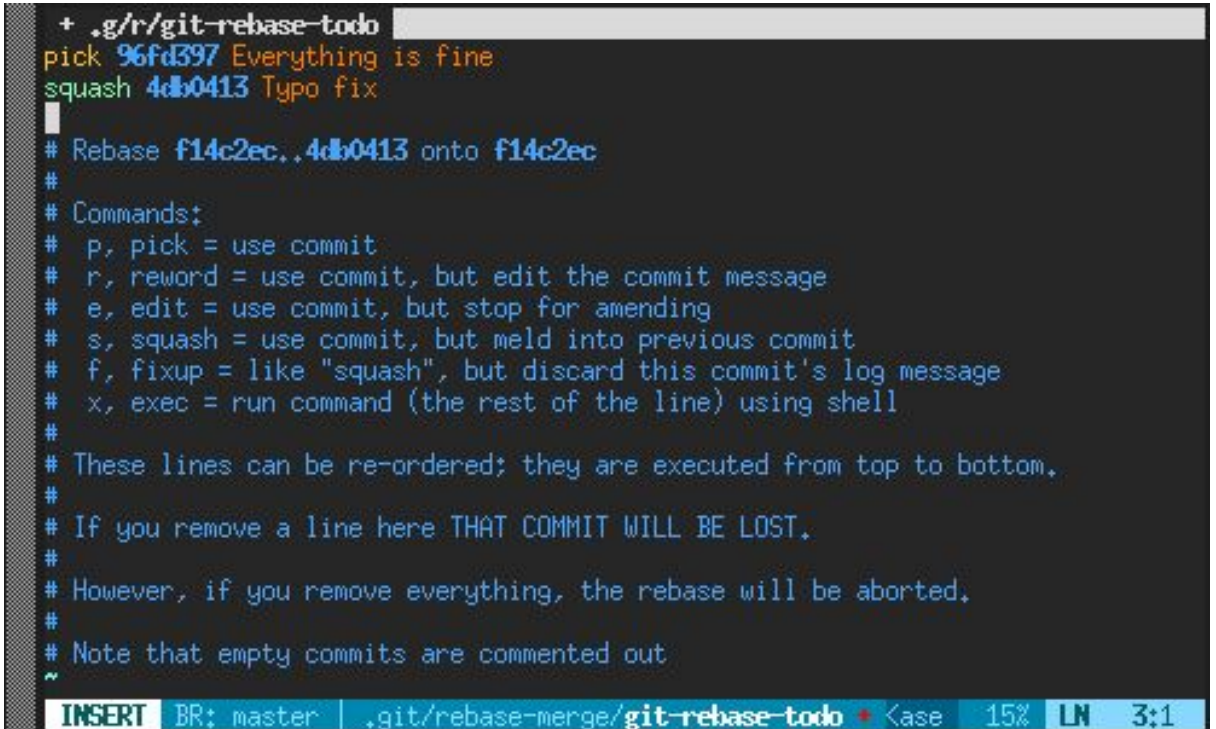
Let’s rebase

Demo$ git rebase -i HEAD^^



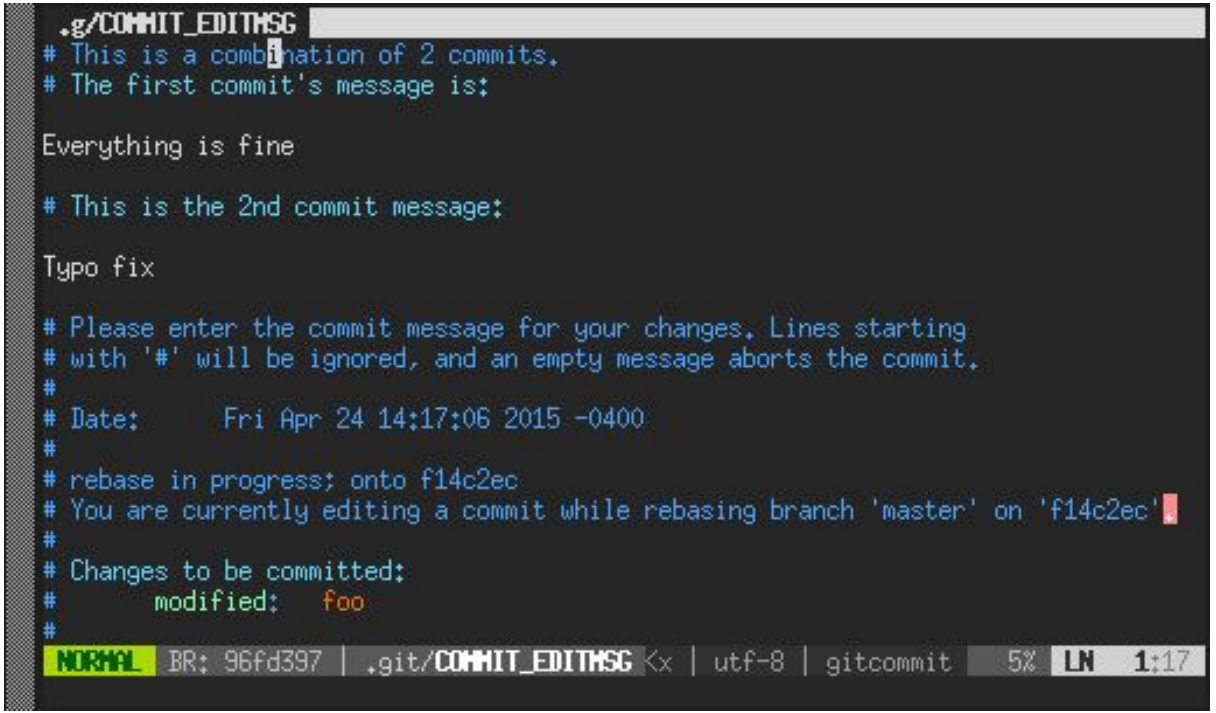
Let’s rebase

Demo$ git rebase -i HEAD^^



Squashing

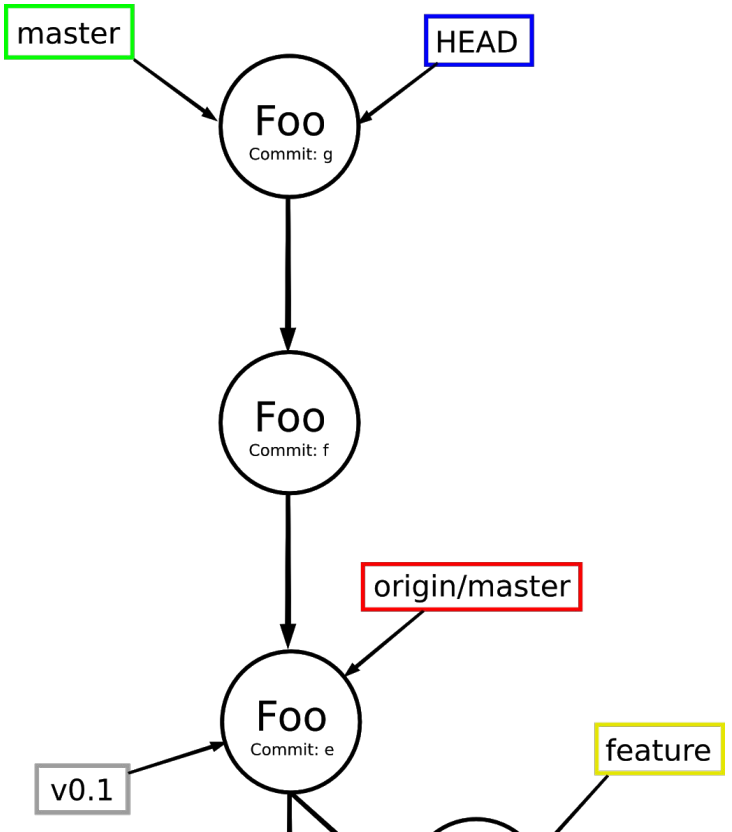
(Other rebase operations are available)



What does this mean for DAG?

f: Introduce error

g: bugfix commit



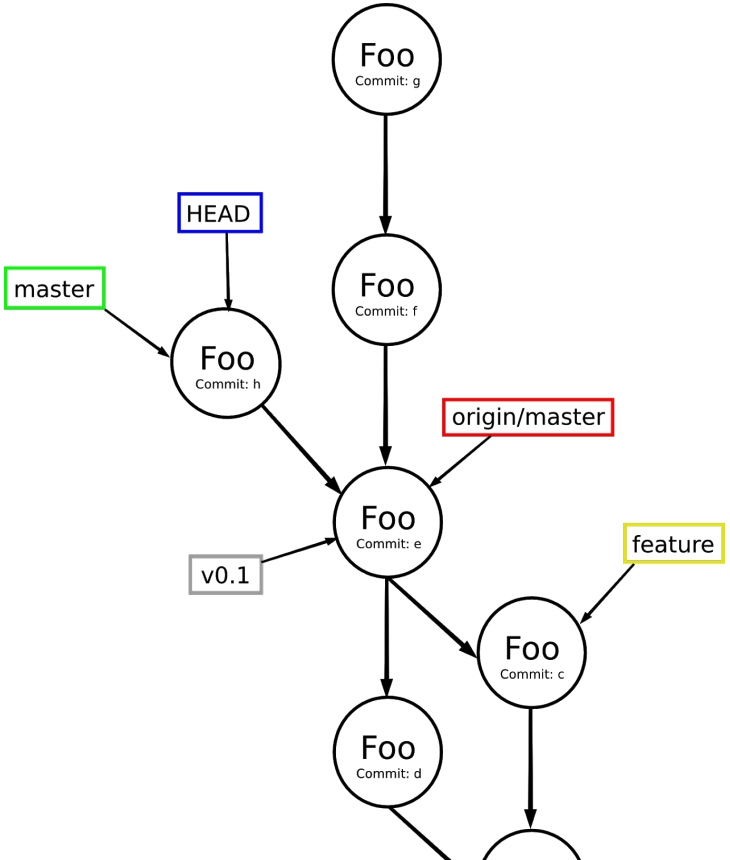
What does this mean for DAG?

f: Introduce error

g: Bugfix commit

-- rebase --

h: Squashed commit

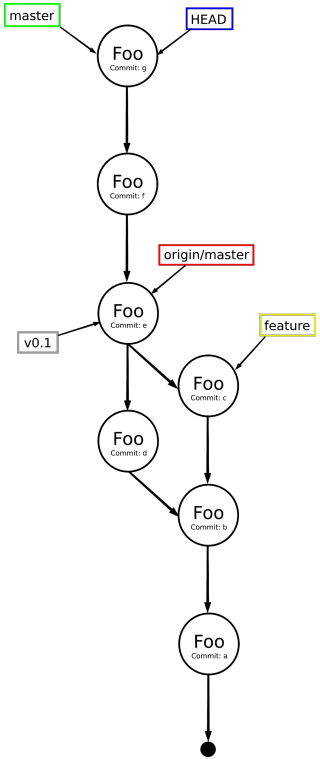


Cool, glad we sorted that

```
Demo$ git push
To ../Remote/
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to '../Remote/'
```

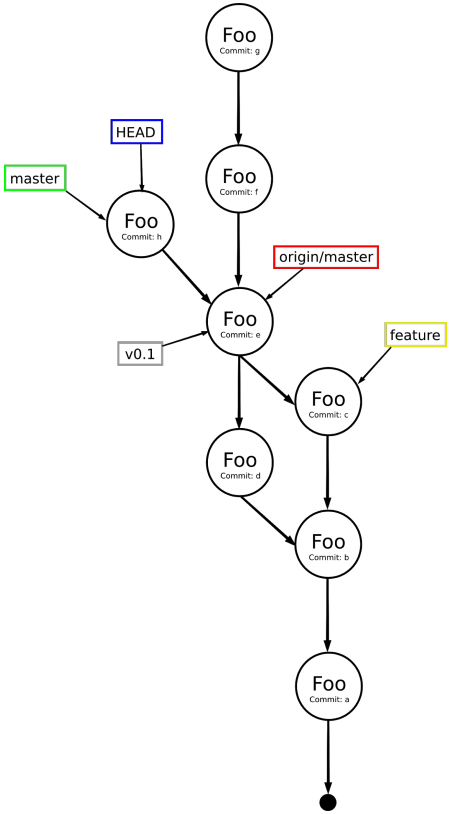
What have we wrought

What went wrong?



Remote

`git push origin master?`



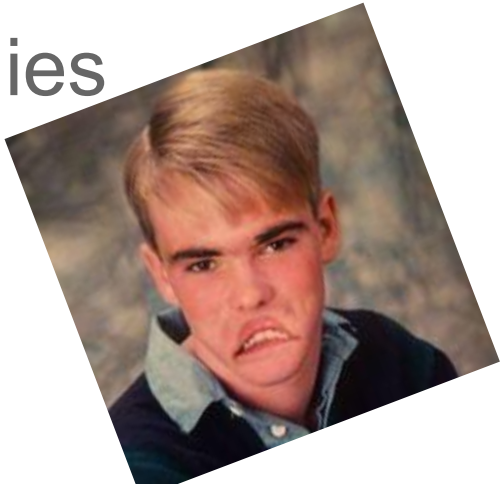
Local

NEVER rebase pushed code

This is how you make enemies

NEVER rebase pushed code

This is how you make enemies



Fetch

Pull refs (labels) from a remote

Find out what others have done

Pull

Pull = fetch + merge

(by default)

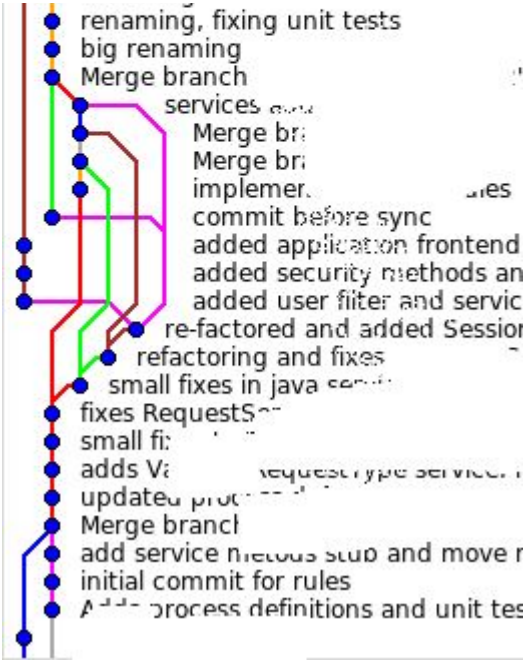
If you’re lucky, it’s a fast-forward

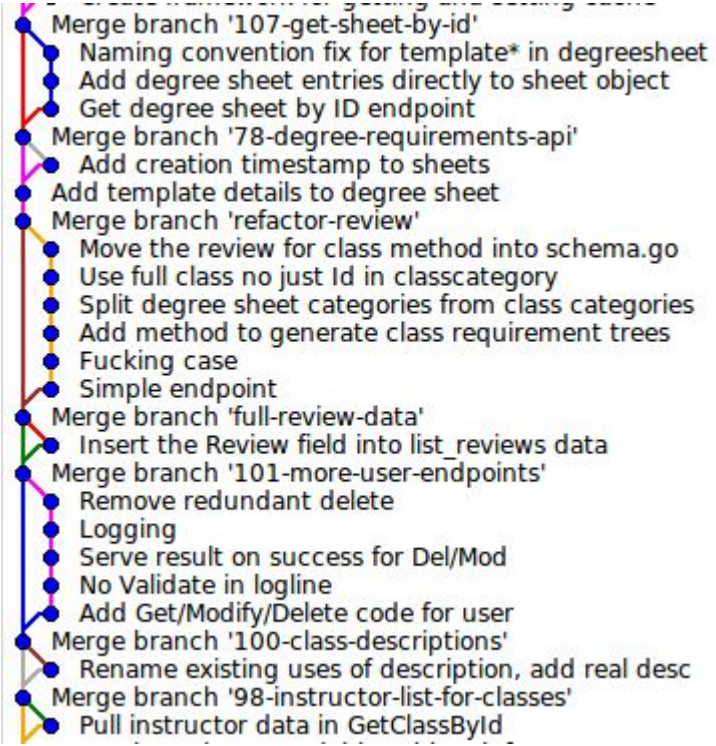
If you have more than one developer, it’s not

(and that’s bad)

Word of caution

Merging many branches all the time is a mess





How to avoid?

Git pull will merge by default

But there’s another behaviour...

How to avoid?

Git pull will merge by default

But there’s another behaviour…

~ rebase ~

The situation:

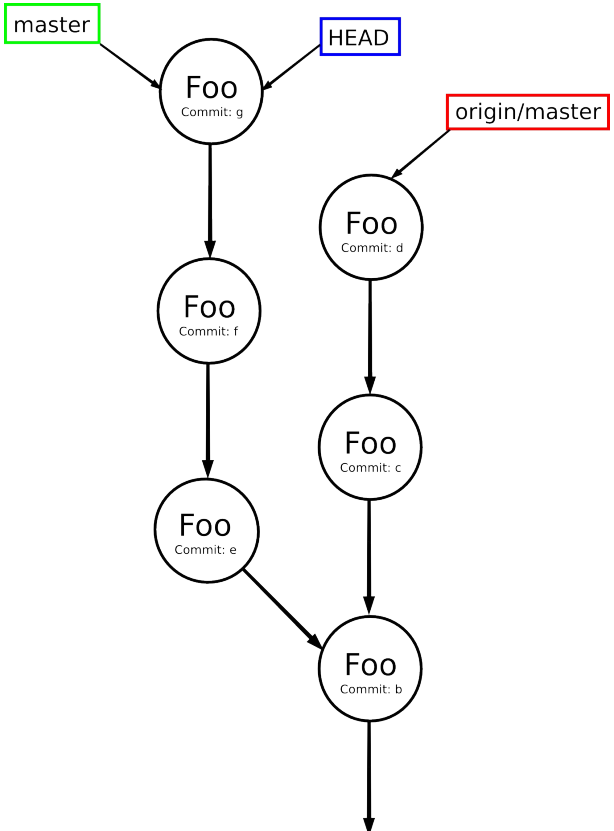
You are working on a branch

Someone else works on that branch, or master

They push, changing either master or branch

- You can’t just push (You’re behind!)
- You don’t want to merge (It’s messy!)

The Situation:



The solution

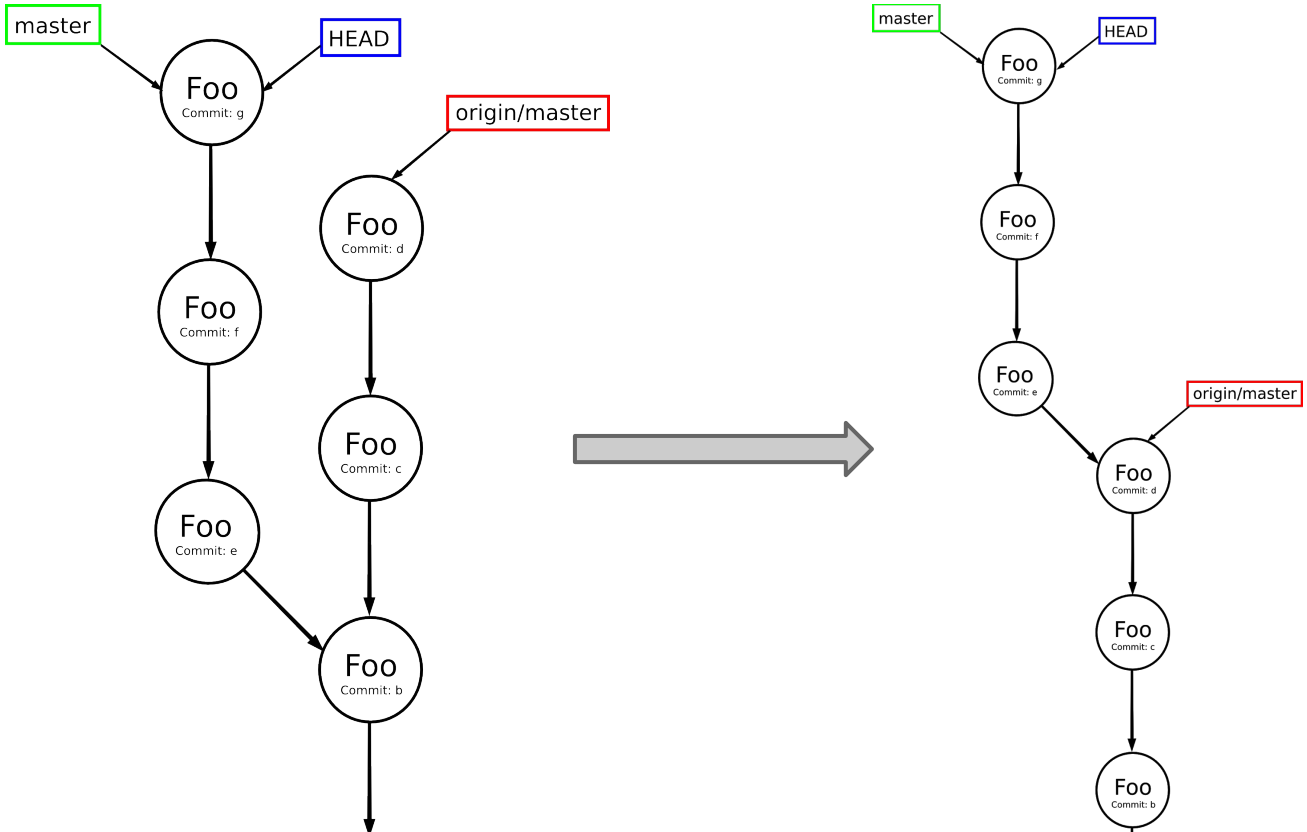
If the branch I am working on changed:

```
Demo$ git pull --rebase
(which is sugar for)
Demo$ git fetch --all
Demo$ git rebase origin/branch
```

If the branch I am branched off of changed:

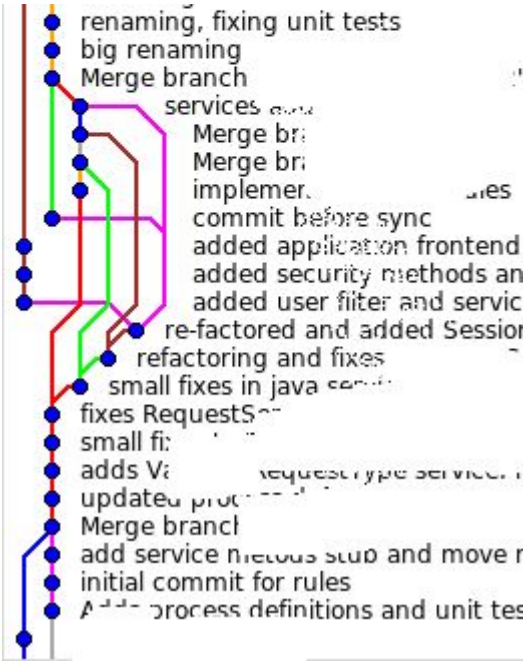
```
Demo$ git rebase master
```

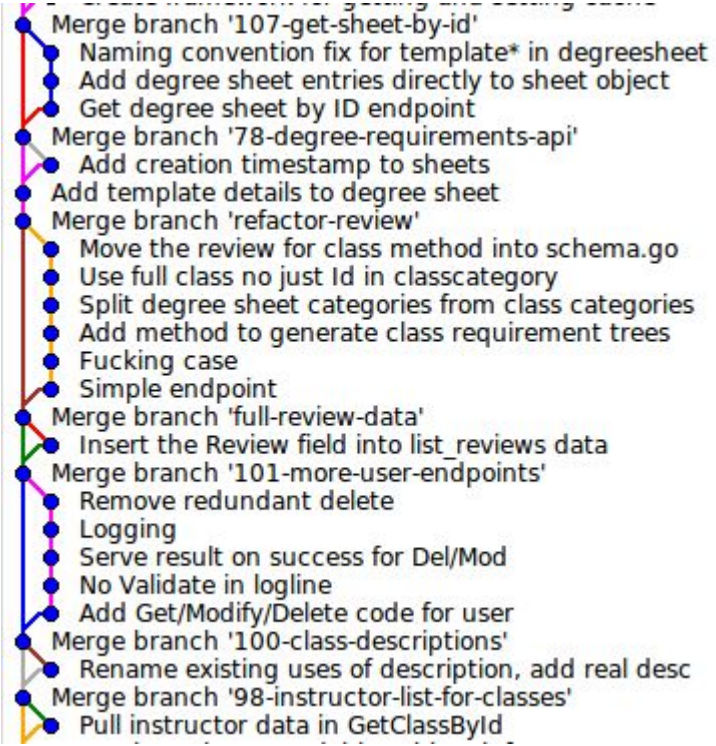
What does this look like?



That’s the big secret

Rebase = the clean history you hear about





One last thing: reflog

Or, “How to recover from anything”
Git keeps track of all commits it’s seen lately

Even if they are no longer referenced

```
52263d9 HEAD@{4}: rebase -i (finish): returning to
refs/heads/master
52263d9 HEAD@{5}: rebase -i (squash): Everything is fine
96fd397 HEAD@{6}: rebase -i (start): checkout HEAD^^
4db0413 HEAD@{7}: commit: Typo fix
```

That’s all folks

Questions? Comments? Obscenities?