Introduction to Git

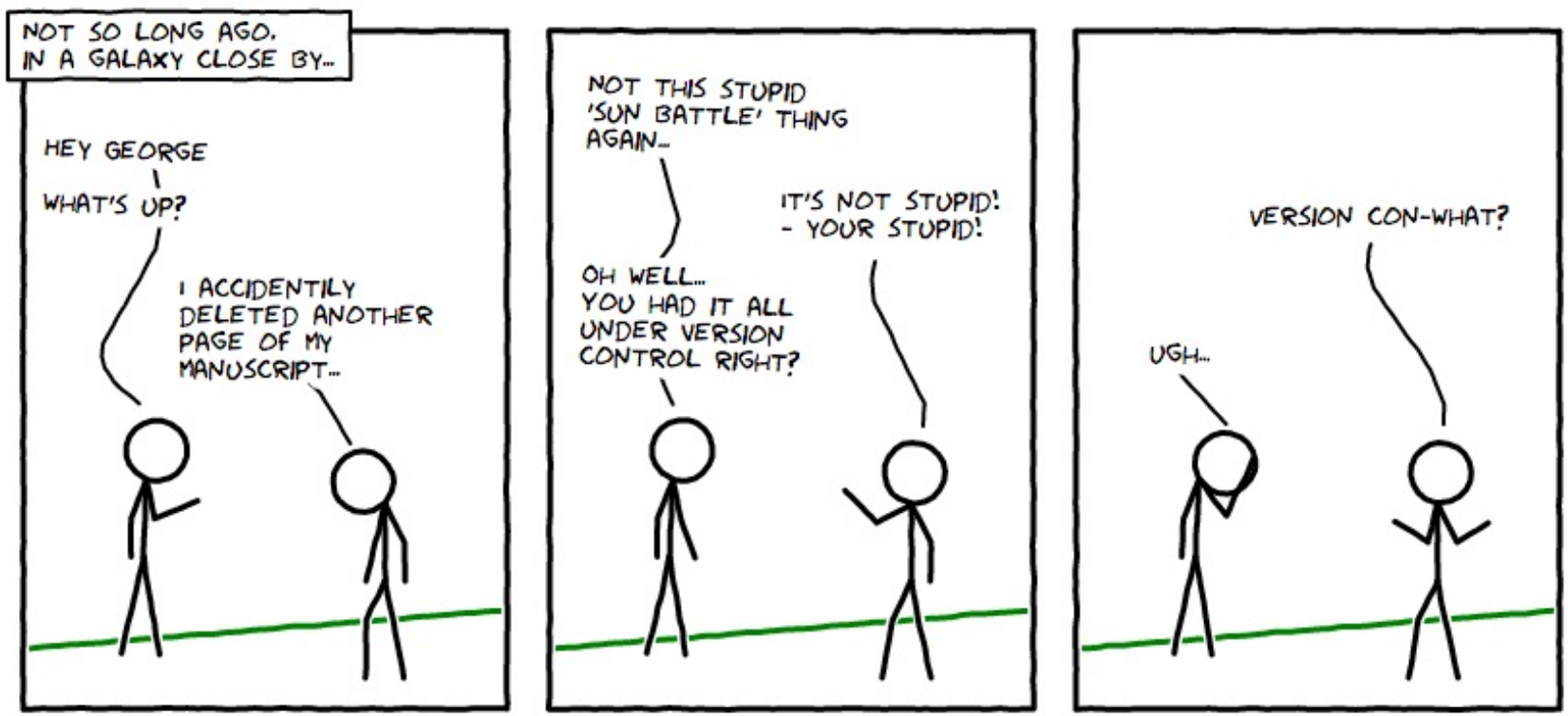
Ali Komaty<akomaty@gmail.com>

March 26, 2020

Outline

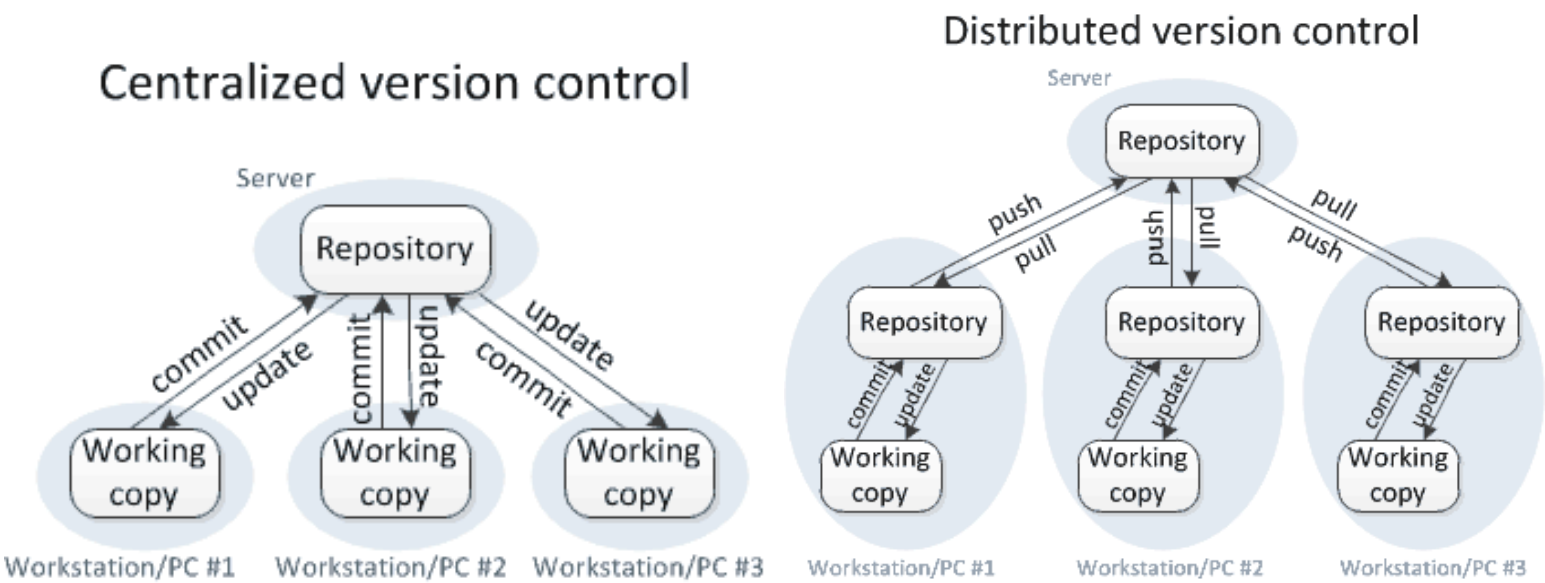
Outline

Version Control



What is Revision Control?

- Management of changes to documents/code or any sorts of collections of information
- It is normally done by specialized software packages such as `git`
- There are two types:
  - Centralized: Revision history is kept on a remote server
  - Distributed: History is copied with the repository



Why is it necessary?

Imagine a world w/o version control:

- You released version 1.0 of your software. It has a bug. Which other versions are affected?
- When was the last time I touched this file? Which changes did I do?
- You introduced a bug on the software: Where is that *fracking* backup?

It is possible!

Actually, the Linux project stayed 11 years w/o version control!

This was possible thanks to an "extremely" organized procedure for diff/patching changes that gave birth to what is "Git" today!

About Version Control

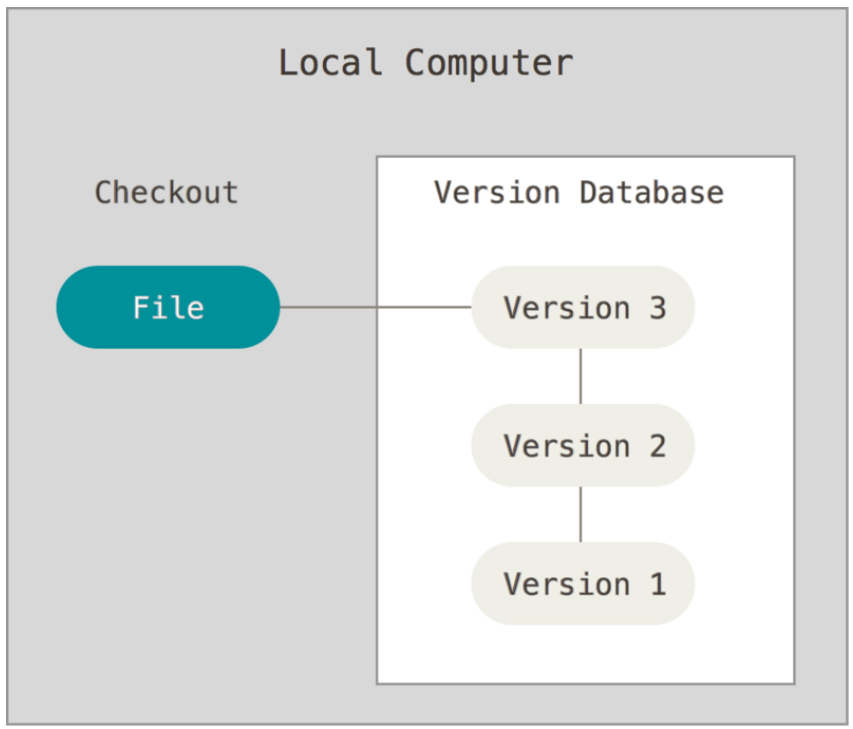
If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to:

- revert selected files back to a previous state,
- revert the entire project back to a previous state,
- compare changes over time,
- see who last modified something that might be causing a problem,
- who introduced an issue and when, and more.

Outline

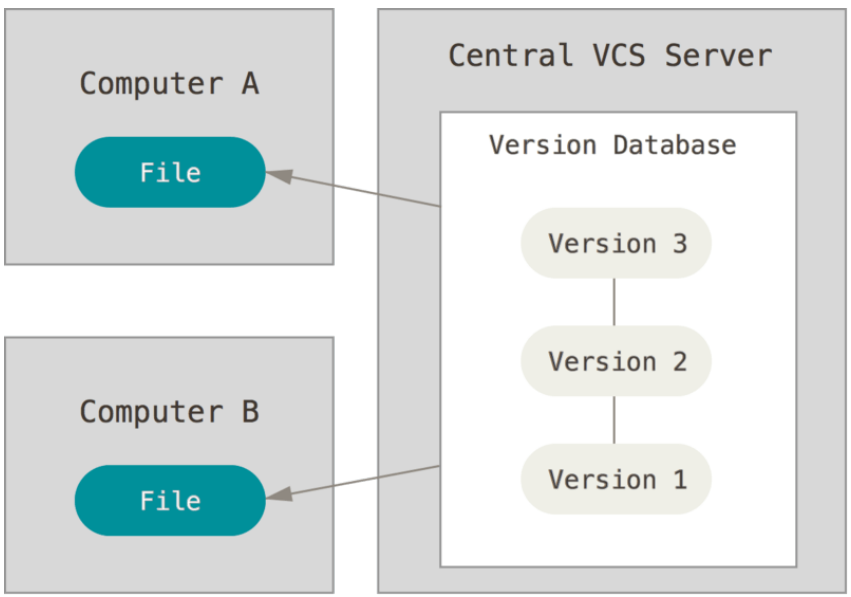
Local Version Control Systems

One of the more popular VCS tools was a system called RCS, which is still distributed with many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.



Centralized Version Control Systems (1)

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place.



Centralized Version Control Systems (2)

This setup offers many advantages, especially over local VCSs.

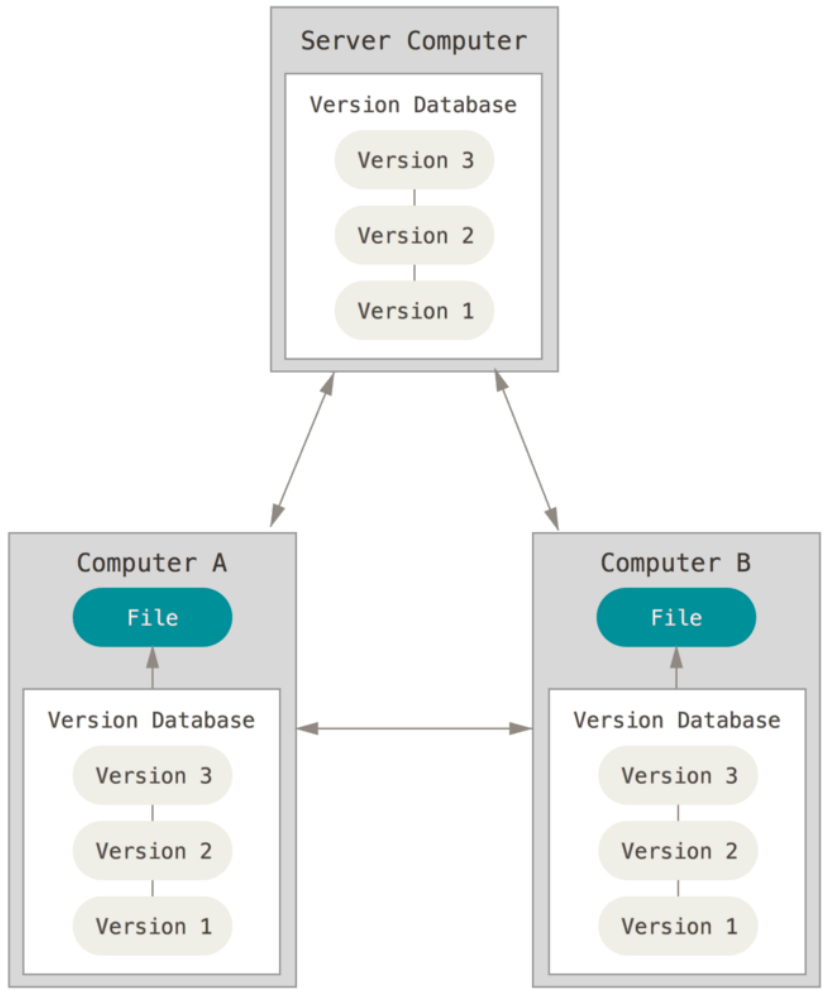
- everyone knows to a certain degree what everyone else on the project is doing.
- Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides:

- If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.
- If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything-the entire history of the project.

Distributed Version Control Systems

In a DVCS, such as Git, clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history.



Distributed Version Control Systems

If any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data. Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

Outline

Installing git

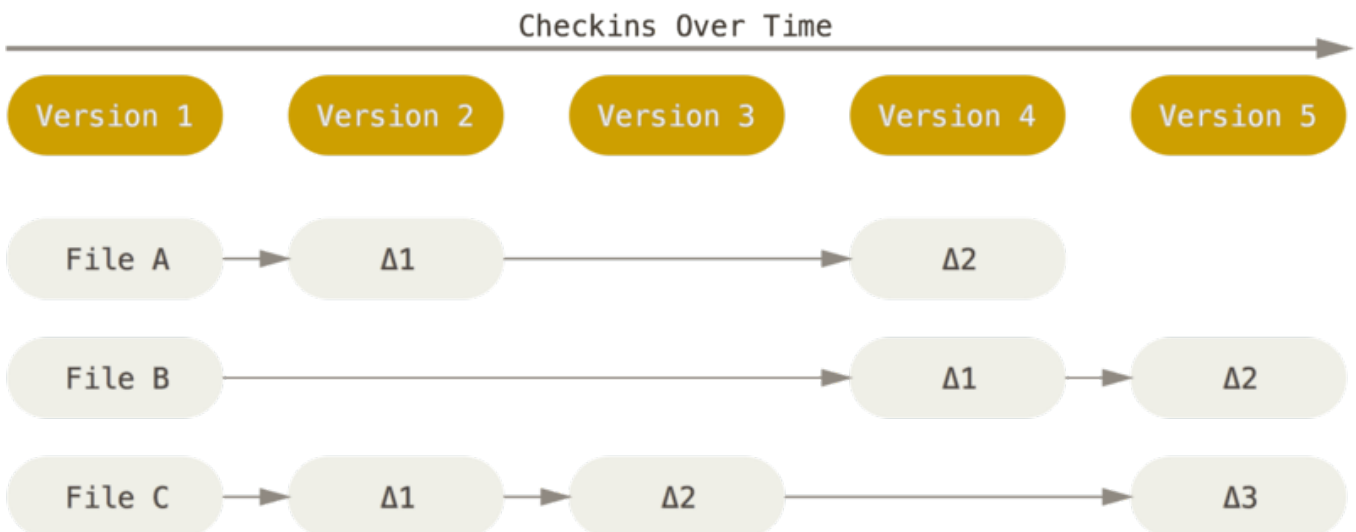
The first step on the way to using Git is to install it! The directions found in the Git documentation below are pretty thorough and helpful, check them out for the best method of getting Git onto your platform of choice.

- Git download page
- Git installation instructions for each platform

Outline

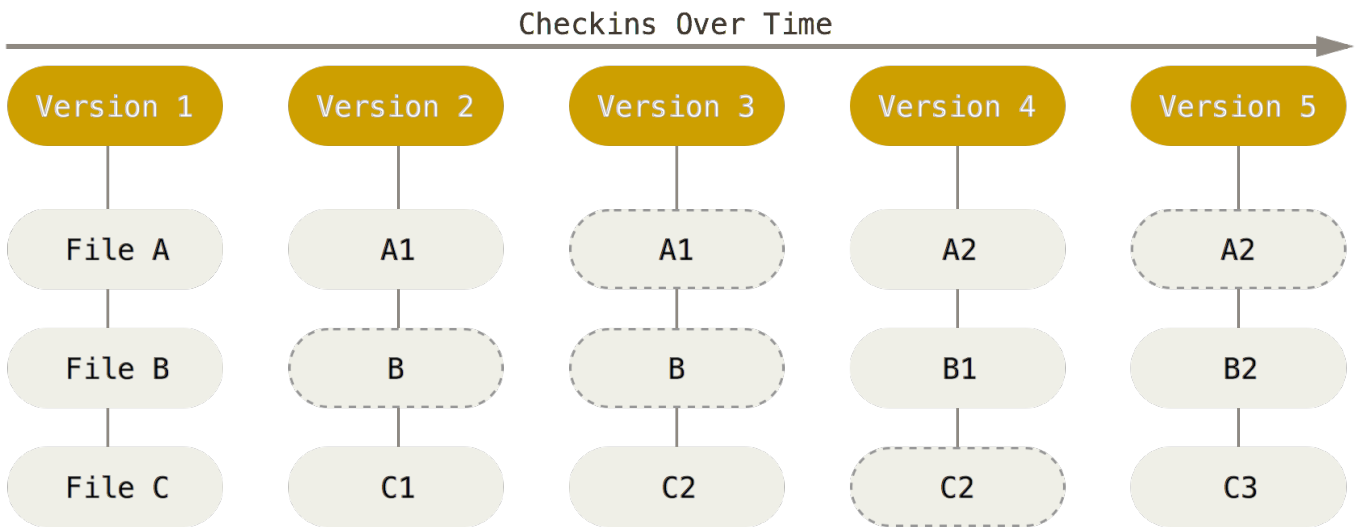
Snapshots, Not Differences

- The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data.
- most other systems store information as a list of file-based changes.



Snapshots, Not Differences

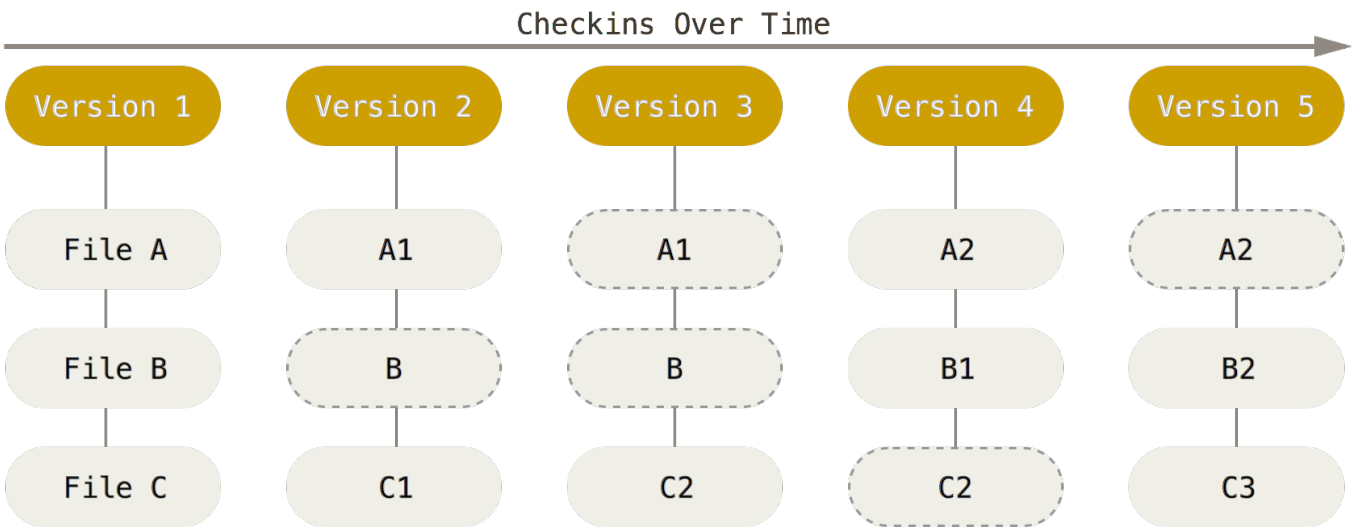
- Git doesn't think of or store its data this way.
- Git thinks of its data more like a series of snapshots of a miniature filesystem.
- Every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
- To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.
- Git thinks about its data more like a **stream of snapshots**.



Git

What is Git?

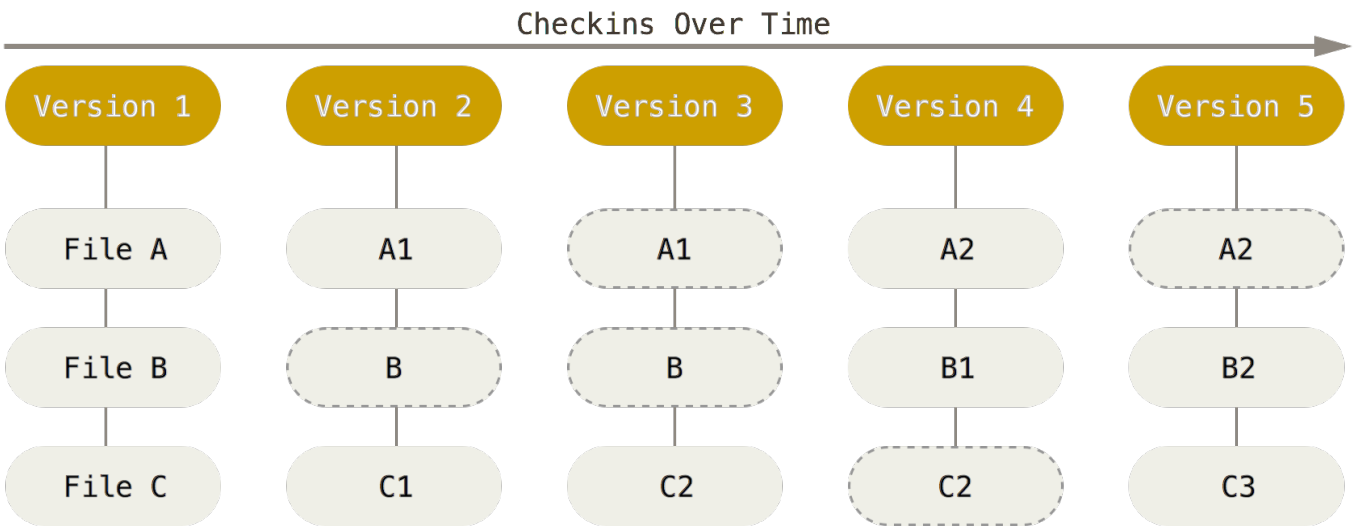
Git is a distributed revision control system. It keeps **snapshots** of **the entirety** of your versioned directory through time using patches.



Git

What is Git?

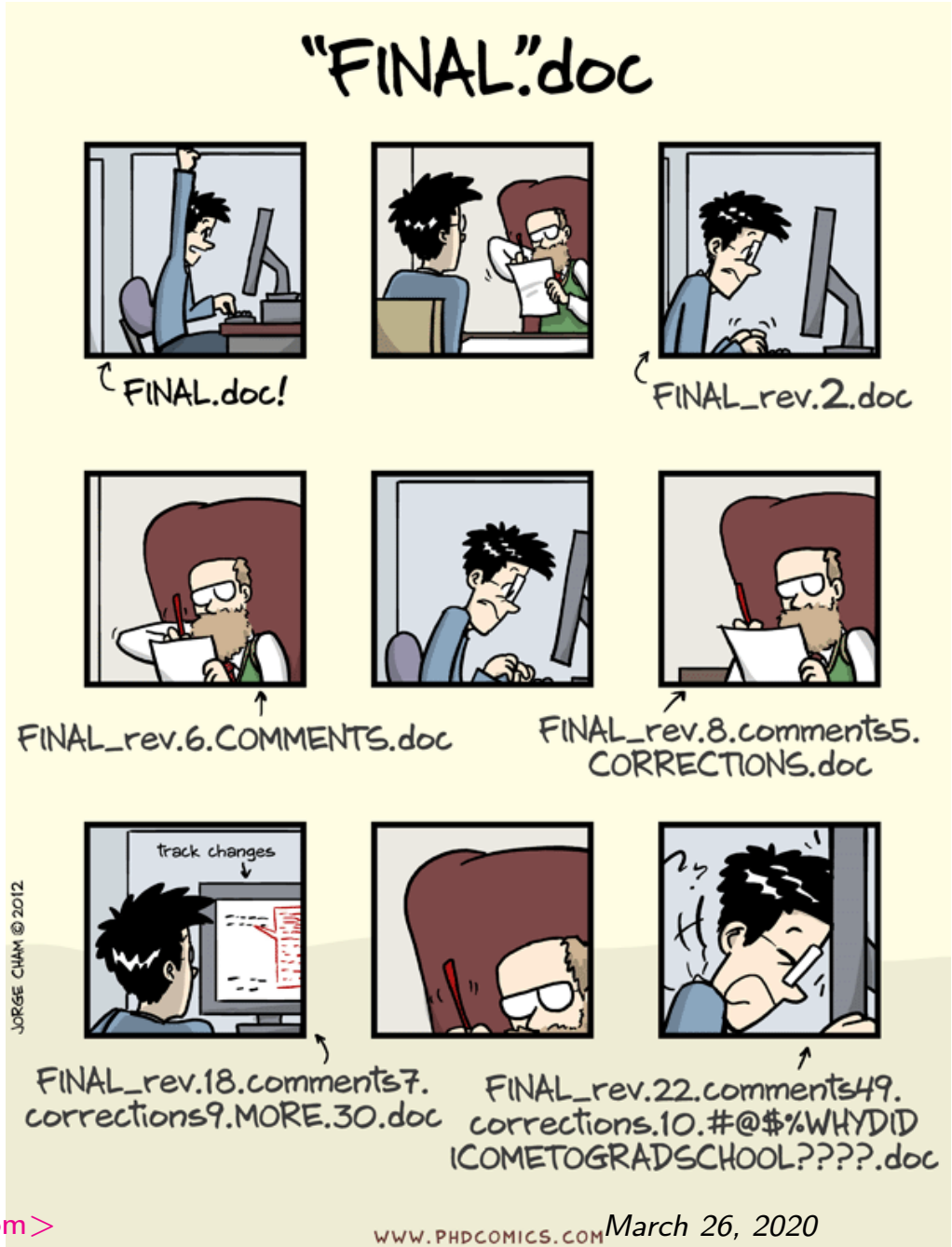
Git is a distributed revision control system. It keeps **snapshots** of **the entirety** of your versioned directory through time using patches.



Old tools, new usage

In order to create a snapshot, git uses *diffs*, *patches* and (SHA-1) *hashes*

$h!tty situation!



Diff/Patch

A *diff* is a set of textual differences between files.

`file1.txt:`

```
I need to go to the store.
I need to buy some apples.
When I get home, I'll wash the dog.
```

`file2.txt:`

```
I need to go to the store.
I need to buy some apples.
I also need to buy grated cheese.
When I get home, I'll wash the dog.
```

To create a *patch*, use the `diff` command:

```
$ diff file1.txt file2.txt > patch.txt
$ cat patch.txt
2a3
> I also need to buy grated cheese.
```

Translating

After line 2 in the first file, a line needs to be added: line 3 from the second file.

Applying patches (1)

Say you're receiving a *diff* file, and you want to apply the changes. Do you know how to do it?

- You can do it manually (but what's the point?)
- You can use the command *patch*

Let's say you wrote the following code:

```
#!/usr/bin/env python
import psutil

def check_cpu_usage(percent):
    usage = psutil.cpu_percent()
    return usage > percent

if not check_cpu_usage(75):
    print("ERROR! CPU is overloaded")
else:
    print("Everything is ok")
```

Then you noticed that there's something not correct so you asked a friend to help you!

Applying patches (2)

Your friend sent you the following *patch* file named *cpu_usage.diff*:

```
--- cpu_usage.py          2020-03-25 17:58:08.194188841 +0200
+++ cpu_usage1.py         2020-03-25 18:05:50.529974766 +0200
@@ -2,7 +2,8 @@
 import psutil

 def check_cpu_usage(percent):
-    usage = psutil.cpu_percent()
+    usage = psutil.cpu_percent(1)
+    print("DEBUG: usage: {}".format(usage))
     return usage > percent

 if not check_cpu_usage(75):
```

Do you notice what are the changes that your friend made? How to integrate them in your code?

Applying patches (3)

To include the changes into your code *cpu_usage.py*, you can simply use the following command:

```
patch cpu_usage.py < cpu_usage.diff
```

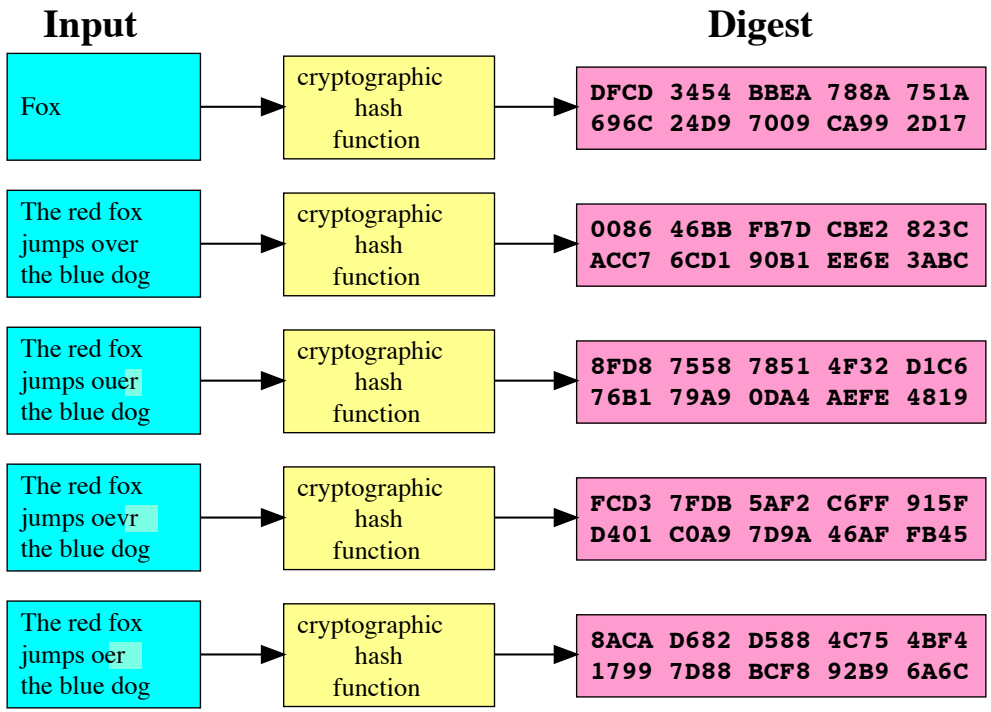
Then your code will be as follows:

```
#!/usr/bin/env python
import psutil

def check_cpu_usage(percent):
    usage = psutil.cpu_percent(1)
    print("DEBUG: usage: {}".format(usage))
    return usage > percent

if not check_cpu_usage(75):
    print("ERROR! CPU is overloaded")
else:
    print("Everything is ok")
```

Outline

Hash

A (crypto) hash function is a function that can be used to map digital data of arbitrary size to a fixed length string, that is practically impossible to invert.



Notice that small changes on the input make the hash change a lot.

Hash (collision)

Nearly impossible to clash

It is nearly **impossible** that two natural sequences collide on the **same** repository.

If all world population would be developers and every one of them would commit to the **same** repository every second, the probability of 50% collision would be reached in[1]:

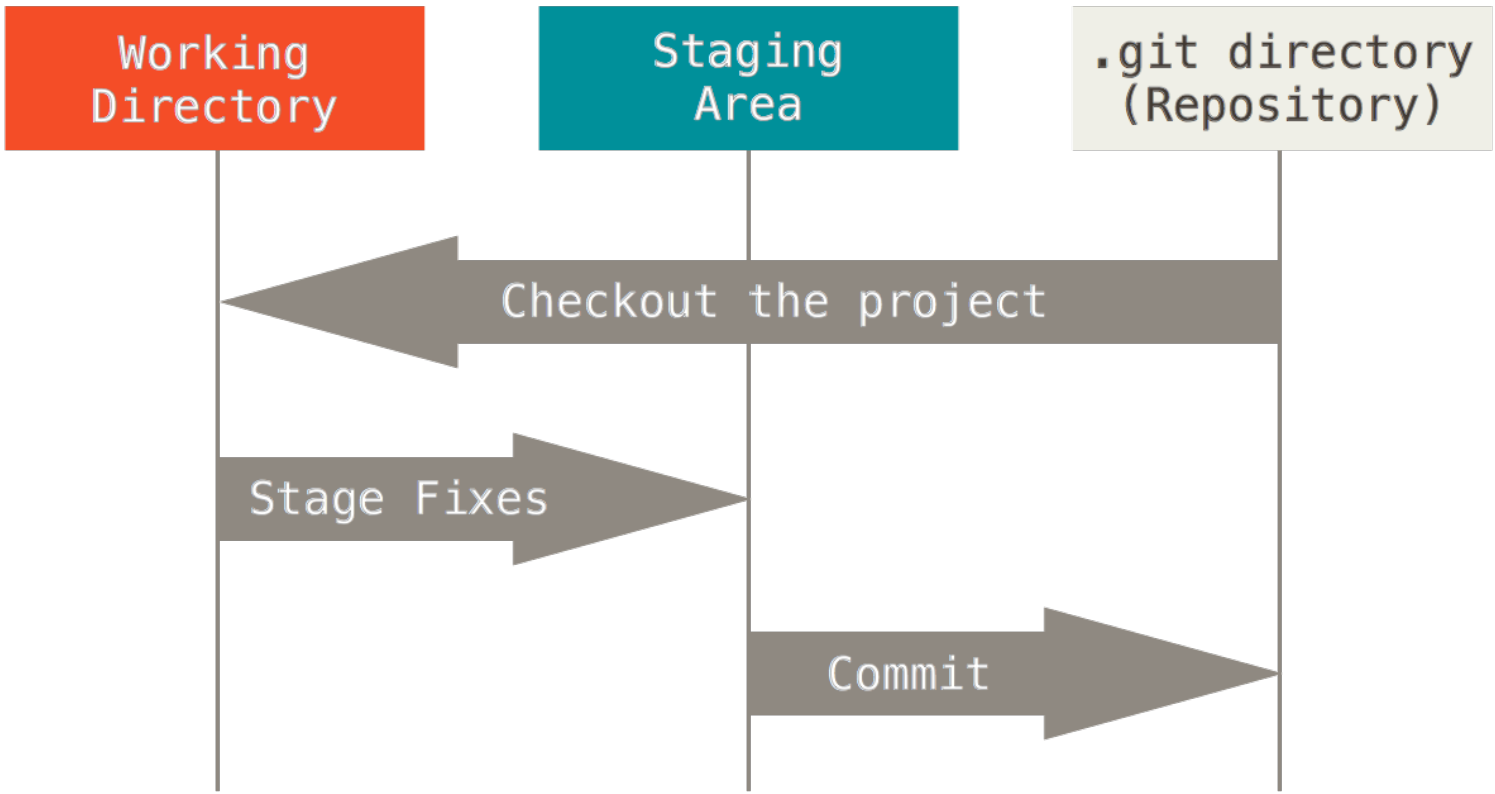
$$6.6 \times 10^6 \text{years}$$

[1] http://diego.assencio.com/?index=eacd6eedf46c9dd596a5f12221ad15b8

Outline

Git states

Git contains 3 states for your project.



Git workflow

Easy

- You modify files in your working directory.
- You stage the files, adding snapshots of them to your staging area.
- You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Outline

Configuring `git` (first time only)

To tell git it should log every commit using your name and e-mail, you need to configure it once:

```
$ git config --global user.name "First Last"
$ git config --global user.email first.last@example.com
# to list all configuration set for you
$ git config --list
```

Tip

Tab-like auto-completion works out-of-the-box!

Initializing a new repository

To initialize a repository just use `git init`. Let's try it!

```
$ cd myproj
$ git init
Initialized empty Git repository in ...
```

What is staged?

The `status` command gives an overview of the staging area.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add ...
```

Let's do the first commit

The `commit` command instructs git to register the snapshot (patch) to its `.git` directory.

```
$ git add . #adds all files to staging area
$ git commit -m "My first commit with git"
$ git status
On branch master
nothing to commit, working directory clean
```

Tip: Configuring the default editor

```
$ git config --global core.editor /usr/bin/nano #default
$ git config --global core.editor /usr/bin/gedit
$ git config --global core.editor /usr/bin/vim
$ git config --global core.editor /usr/bin/gvim
```

Outline

Making changes

The power of version control can be shown when you make changes.

```
$ #make some changes
# now we use git to problem for the modification
$ git diff
diff --git a/analysis.py b/analysis.py
index d4d5b3e..2697486 100644
--- a/analysis.py
+++ b/analysis.py
@@ -20,4 +20,4 @@ def CER(prediction, true_labels):
    """

    errors = (prediction != true_labels).sum()
-   return float(errors)/len(prediction)
+   return errors/len(prediction)
```

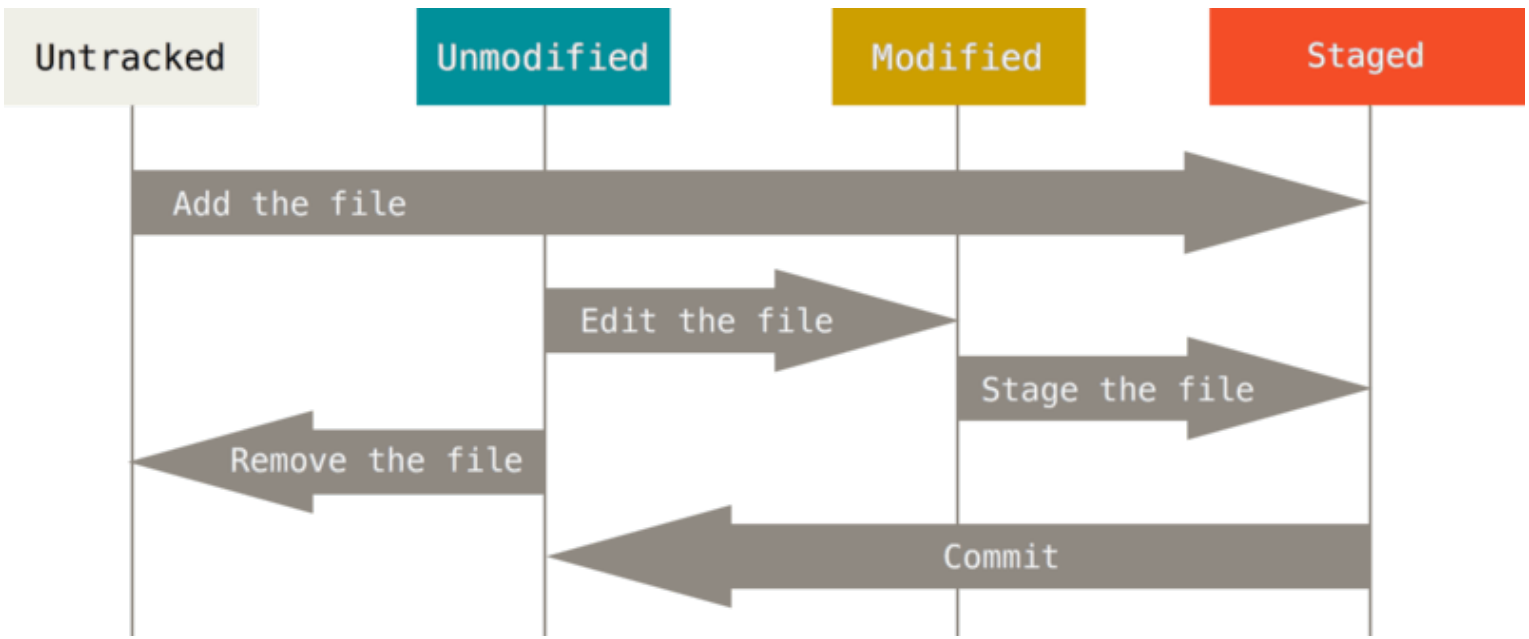
Committing changes (faster)

You can stage and commit changes with one command.

```
$ git commit -m "Re-added nasty bug" -a
```

Recording Changes to the Repository

Each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.



Outline

Logs and Diffs

At all times, you have access to history and can revert back.

```
$ git log --oneline
df7bc06 Re-added nasty bug
ccb2d42 My first commit with git
$ #figured out I re-added the bug, so will revert
$ git revert df7bc06
# edit the comment, and save (<ESC>:wq)
$ git log --oneline
a43f4c6 Revert "Re-added nasty bug"
df7bc06 Re-added nasty bug
ccb2d42 My first commit with git
$ git diff df7bc06..a43f4c6
# OK!
```

Tags

Git allows you to set labels to refer to repository versions (instead of hash initials). You should use the `tag` command to do so.

```
$ git tag final #makes final == a43f4c6..
$ git tag buggy df7bc06 #makes buggy == df7bc06...
$ git diff buggy..final
...
$ git tag
buggy
final
```

Outline

Ignoring Files

Git allows you can create a file listing patterns that you want to ignore.

```
$ cat .gitignore
*.[oa]
*~
```

The first line tells Git to ignore any files ending in ".o" or ".a" The second line tells Git to ignore all files whose names end with a tilde (∼).

Rules for `.gitignore`

The rules for the patterns you can put in the .gitignore file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work, and will be applied recursively throughout the entire working tree.
- You can start patterns with a forward slash (/) to avoid recursivity.
- You can end patterns with a forward slash (/) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (!).

Ignoring Files

Here is another example .gitignore file:

```
# ignore all .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the TODO file in the current directory,
# not subdir/TODO
/TODO
# ignore all files in any directory named build
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .pdf files in the doc/ directory and any of its
# subdirectories
doc/**/*.pdf
```

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the git log command.

Outline

Branches in a Nutshell (1)

- When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged.
- This object also contains the author's name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents)

commit-and-tree

Branches in a Nutshell (2)

- let's assume that you have a directory containing three files, and you stage them all and commit.
- Staging the files computes a checksum for each one (the SHA-1 hash we mentioned before), stores that version of the file in the Git repository (Git refers to them as blobs), and adds that checksum to the staging area:

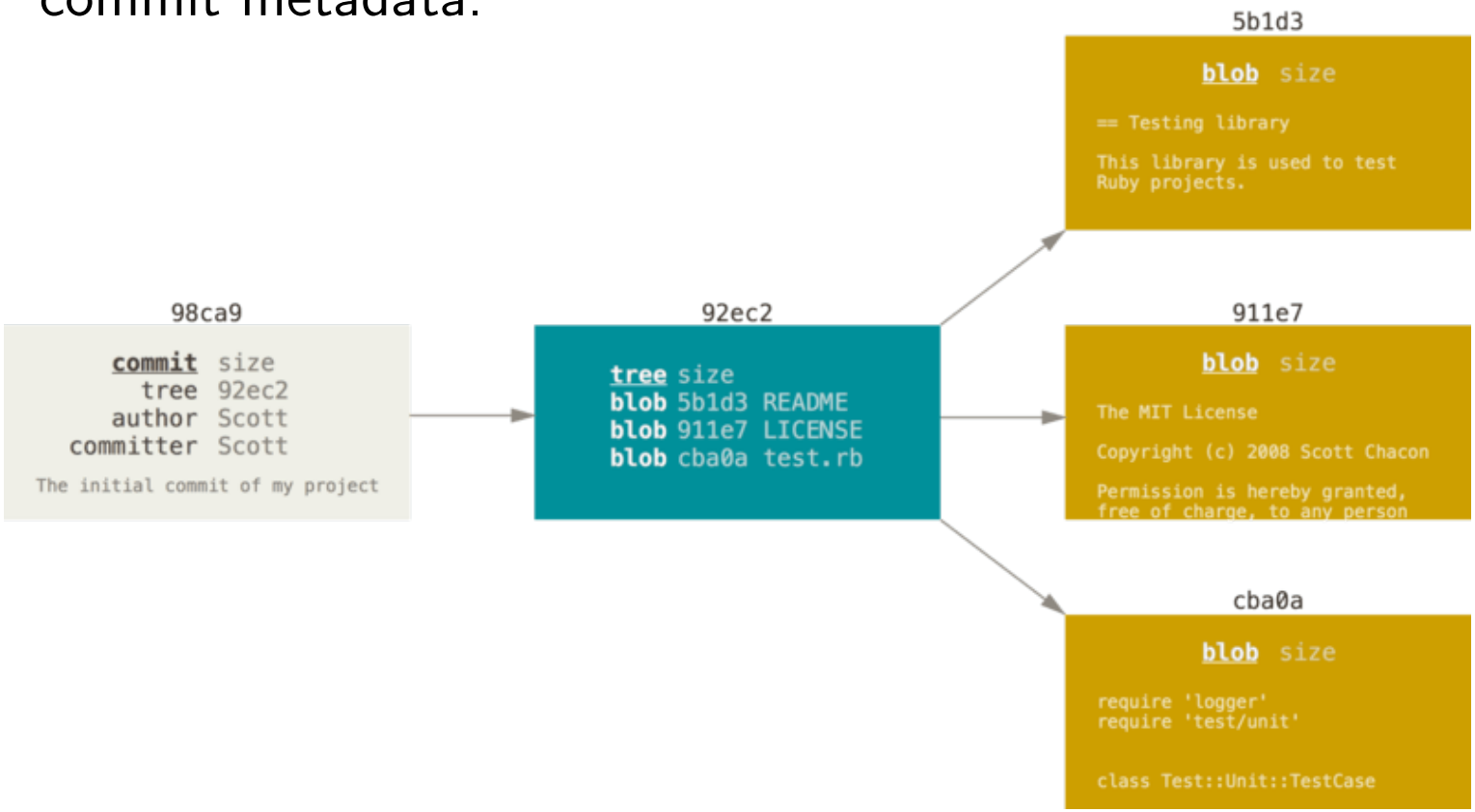
```
git add README test.rb LICENSE
git commit -m 'The initial commit of my project'
```

- When you create the commit by running git commit, Git checksums each subdirectory and stores those tree objects in the Git repository.
- Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Branches in a Nutshell (3)

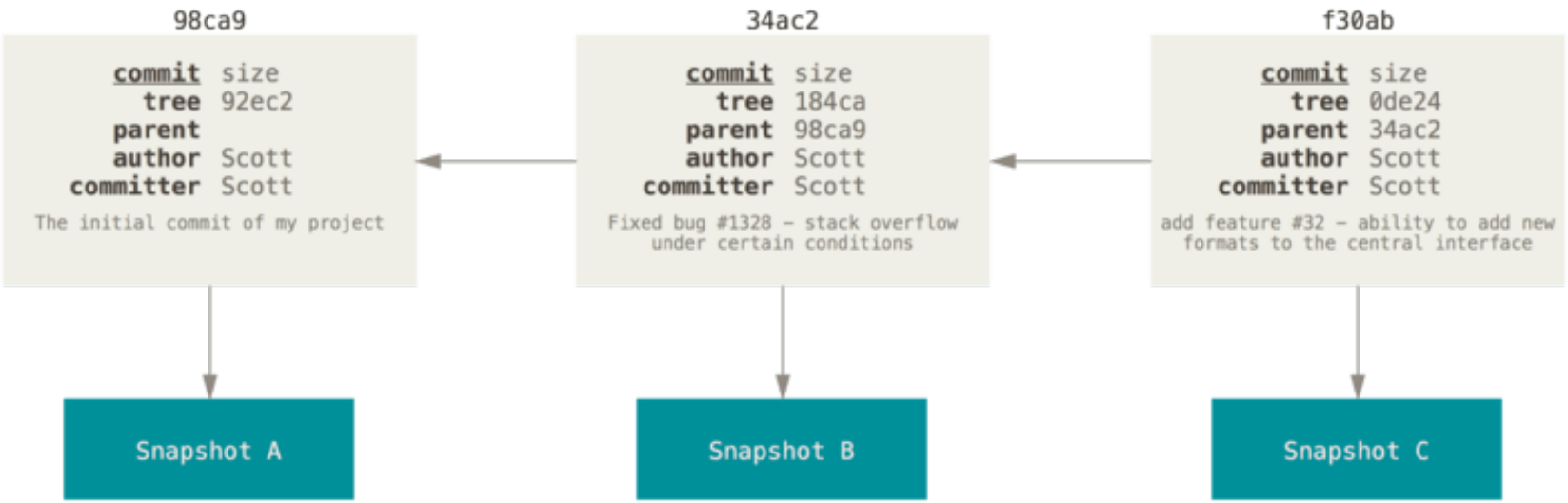
Your Git repository now contains five objects:

- three blobs (representing the contents of one of the three files)
- one tree that lists the contents of the directory and specifies which file names are stored as which blobs
- one commit with the pointer to that root tree and all the commit metadata.



Branches in a Nutshell (4)

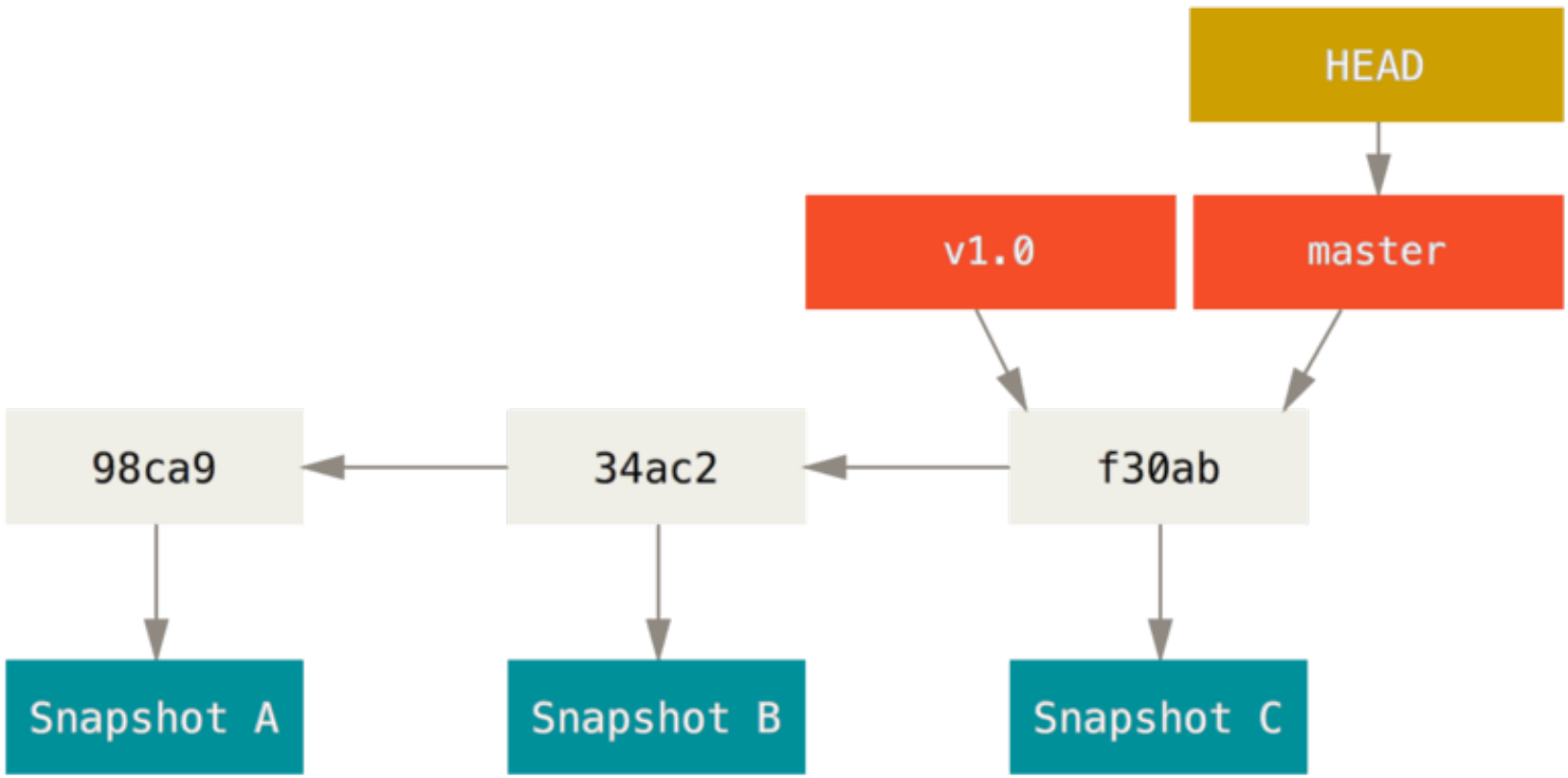
If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



Outline

Branches

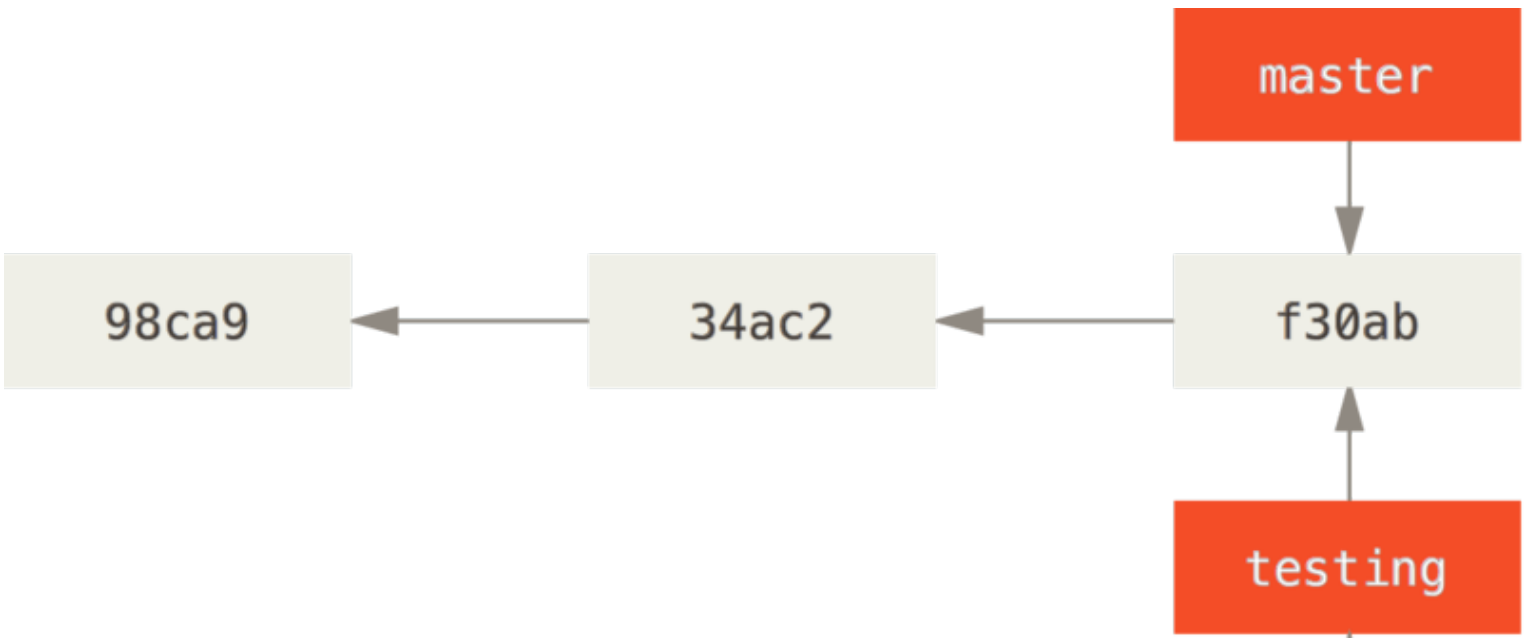
A branch in Git is simply a lightweight movable pointer to one of these commits. By default, there is only one branch (called `master`) on a git repository.



Branches (2)

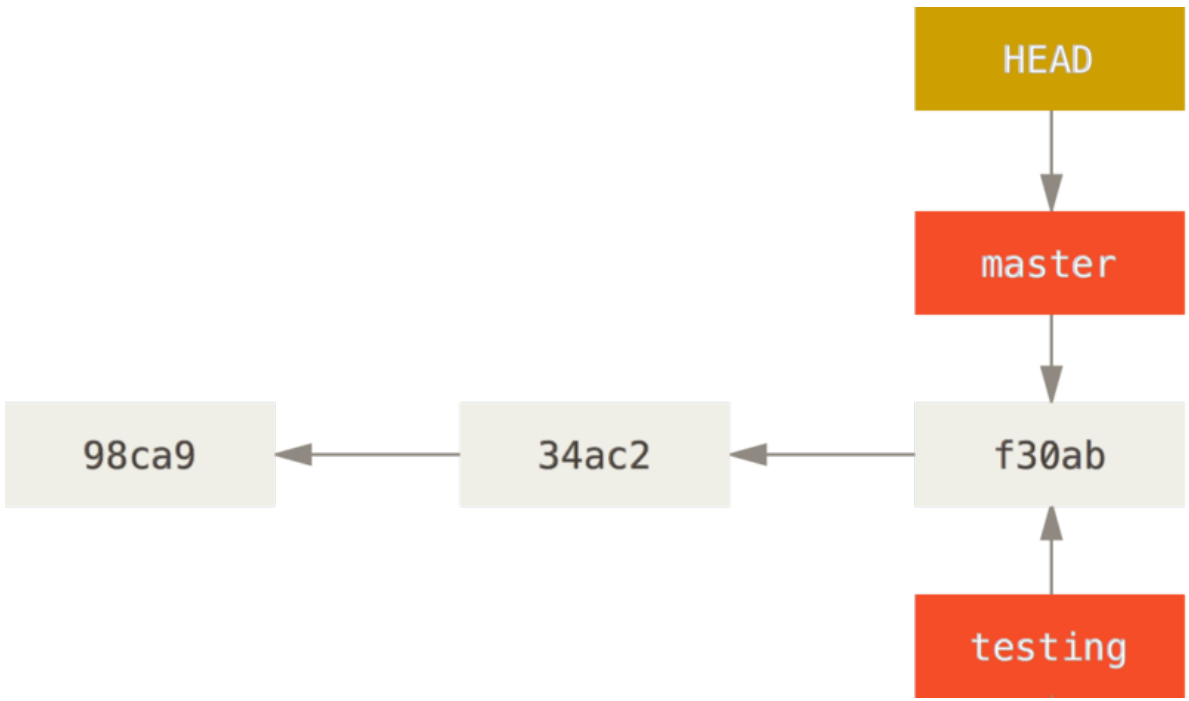
You may create a new branch, to develop something new while keeping the master stable.

```
$ git branch testing
```



Branches (3)

How does Git know what branch you're currently on?
It keeps a special pointer called `HEAD`



Pay attention!

The git branch command only created a new branch - it didn't switch to that branch.

Branches (4)

You can easily see on which branch the `HEAD` is pointing by running a simple `git log` command that shows you where the branch pointers are pointing. This option is called `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) add feature #32 - ability to add
# new formats to the central interface
Fixed bug #1328 - stack overflow under certain conditions
The initial commit of my project
```

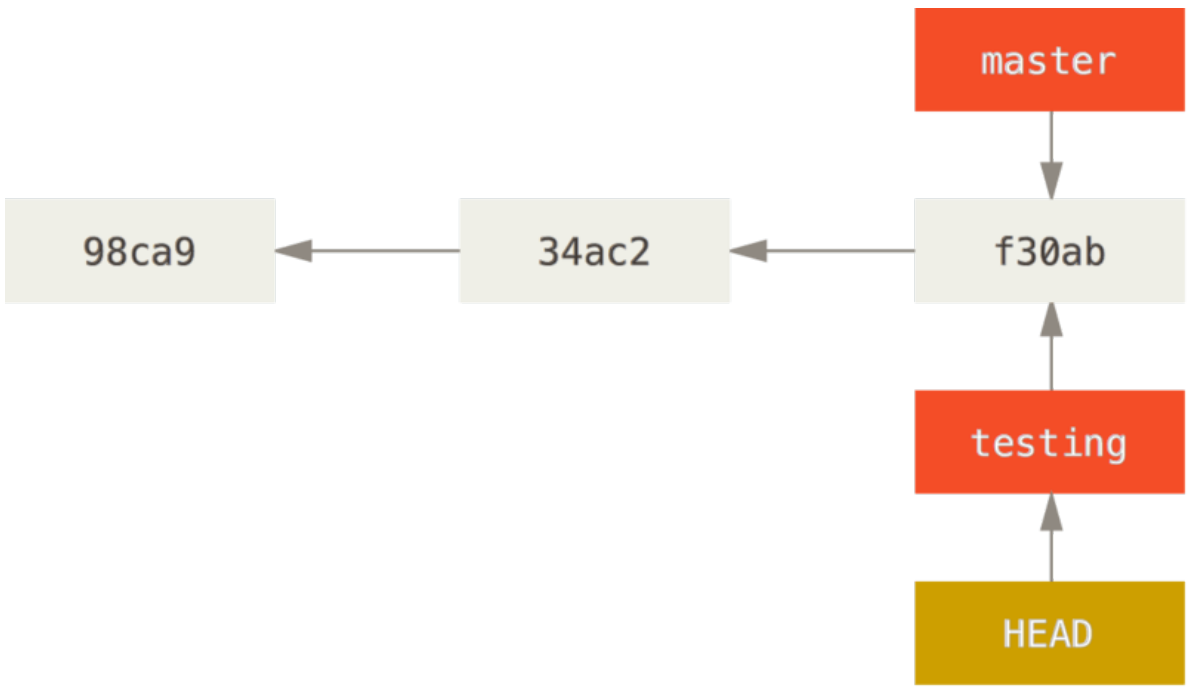
Outline

Switching Branches

To tell Git to consider a new branch as the default for commiting, use the `checkout` command:

```
$ git checkout testing
Switched to a branch "testing"
```

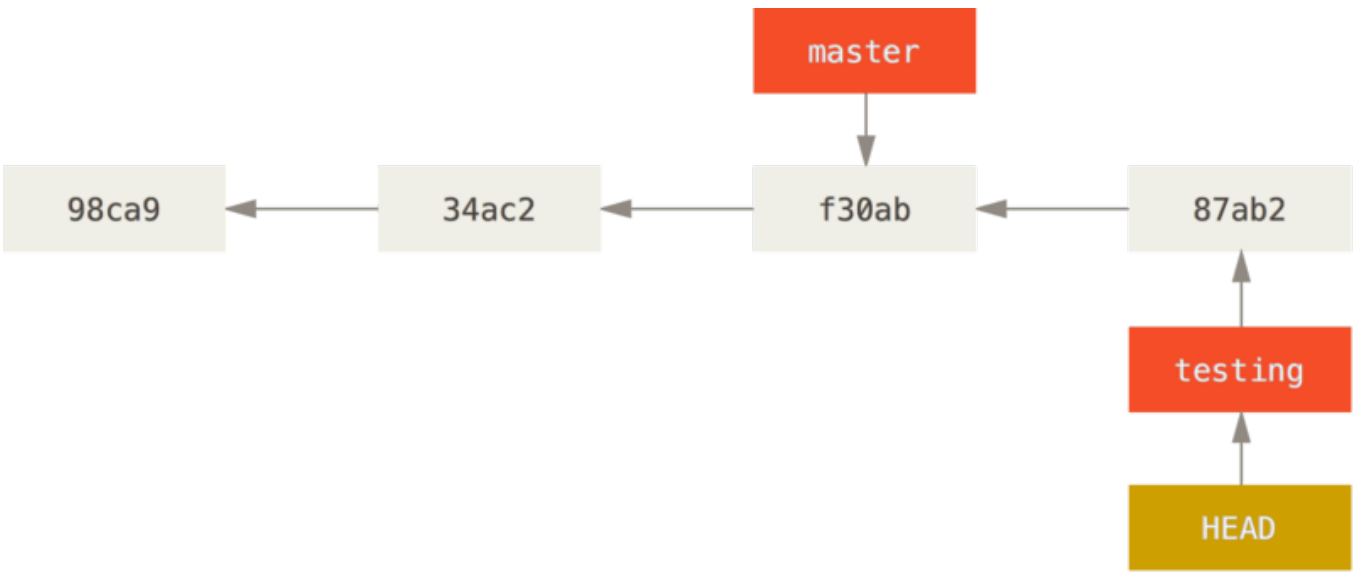
This moves `HEAD` to point to the `testing` branch.



Switching Branches (2)

If you commit a change, only the marker `testing` will be modified:

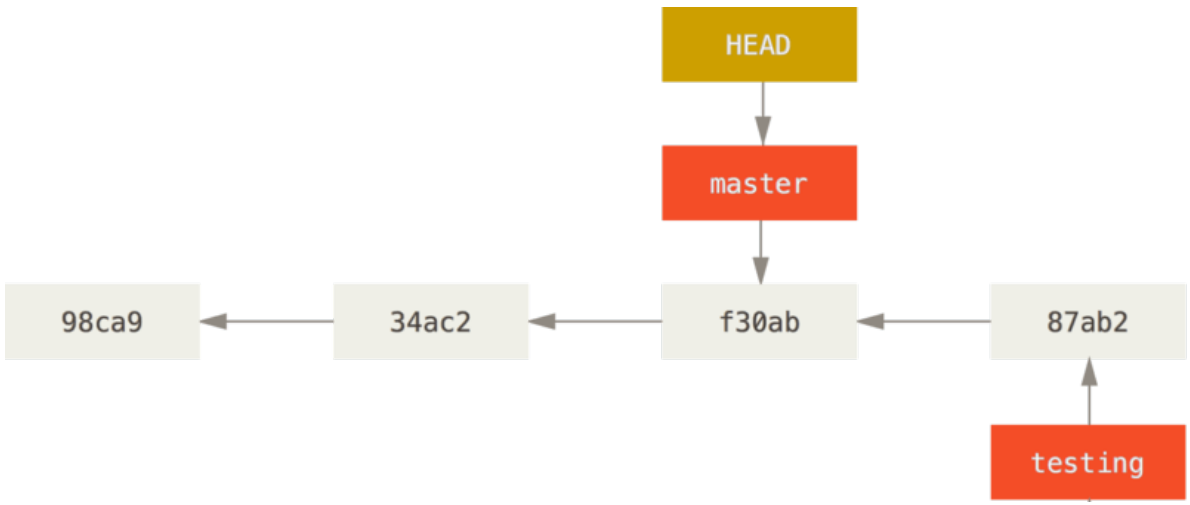
```
$ vim test.rb
$ git commit -a -m 'made a change'
```



Switching Branches (3)

If you commit a change, only the marker `testing` will be modified:

```
$ git checkout master
```



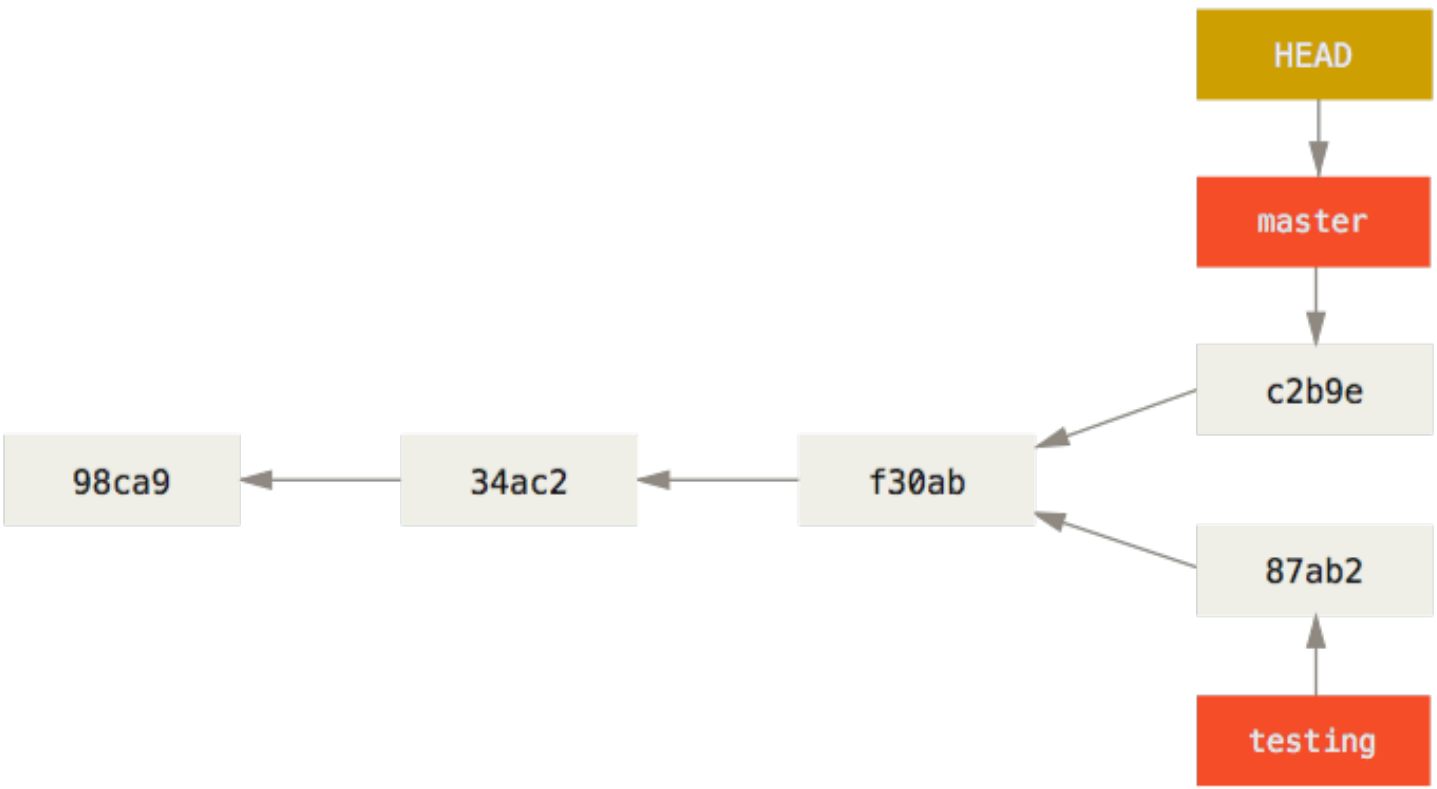
Pay attention!

That command did two things. It moved the `HEAD` pointer back to point to the `master` branch, and it reverted the files in your working directory back to the snapshot that master points to.

Switching Branches (4)

Let's make a few changes and commit again:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```



Switching Branches: some commands

```
$ gedit ...
$ git commit -m "My change to the special branch" -a
$ git log --oneline #has the new commit
...
$ git log --oneline master #as before
...
$ git checkout master #To go back to the master
$ git merge testing #To merge all changes back
$ git branch -d testing #To remove the branch
```

How to Properly Use Tag/Branches

As rule of thumb:

- Everytime you make a **new release**, you **tag** your repository so you know what was distributed
- You use branches to:
  - Test new features w/o disturbing the stability of `master`
  - Fix problems with old versions of the software:

```
$ git branch old-release tag-1.2.4
$ git checkout old-release #state of version 1.2.4
# edit the changes
$ git commit -m "..."
# release version 1.2.5 from that point
```

More Git

Here is a list of resources if you're interested to know more about this powerful tool:

- Reference website: https://git-scm.com/documentation
  - Detailed tutorials
  - Videos
  - Reference documentation

  videos)
- ProGit Book (2nd Edition): http://git-scm.com/book/en/v2
- MOOC: https://www.codeschool.com/courses/try-git
- YouTube clips from GitLab: https://www.youtube.com/channel/UCnMGQ8QHMAnVIsI3xJrihhg

GitLab: Push the code

You can now push your local code to the remote repository.

What to do?

Here is a list of tips:

- **Always** keep your source code under revision control
- **Commit** often
- **Tag** to keep track of important moments (paper status for example)
- Use **Branches** to apply fixes to distributed software
- Use GitLab as much as you can to host your code. It is free, robust and requires no maintenance efforts from you!