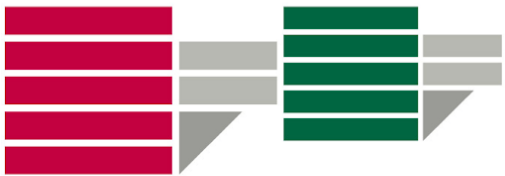


Version Control Systems

UNIVERSITÀ DELLA CALABRIA



DIMES - Dipartimento di INGEGNERIA INFORMATICA
MODELLISTICA, ELETTRONICA E SISTEMISTICA

Ing. Ludovica Sacco
DIMES – UNICAL - 87036 Rende(CS) - Italy
Email: l.sacco@dimes.unical.it

Version Control

About Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

Usually, it is source code being version controlled, though in reality this can be done with nearly any type of file on a computer.

I.E.

You are a graphic or web designer and want to keep every version of an image or layout (which you certainly would), it is very wise to use a Version Control System.

About Version Control

A VCS allows you to:

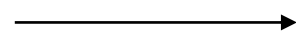
- revert files back to a previous state
- revert the entire project back to a previous state
- review changes made over time
- see who last modified something that might be causing a problem
- who introduced an issue and when
- and more

Using a VCS also means that if you make a mistake or lose files, you can easily recover .

You get all this for very little overhead!

Local Version Control Systems

What is people common choice?
Copy files into another directory!

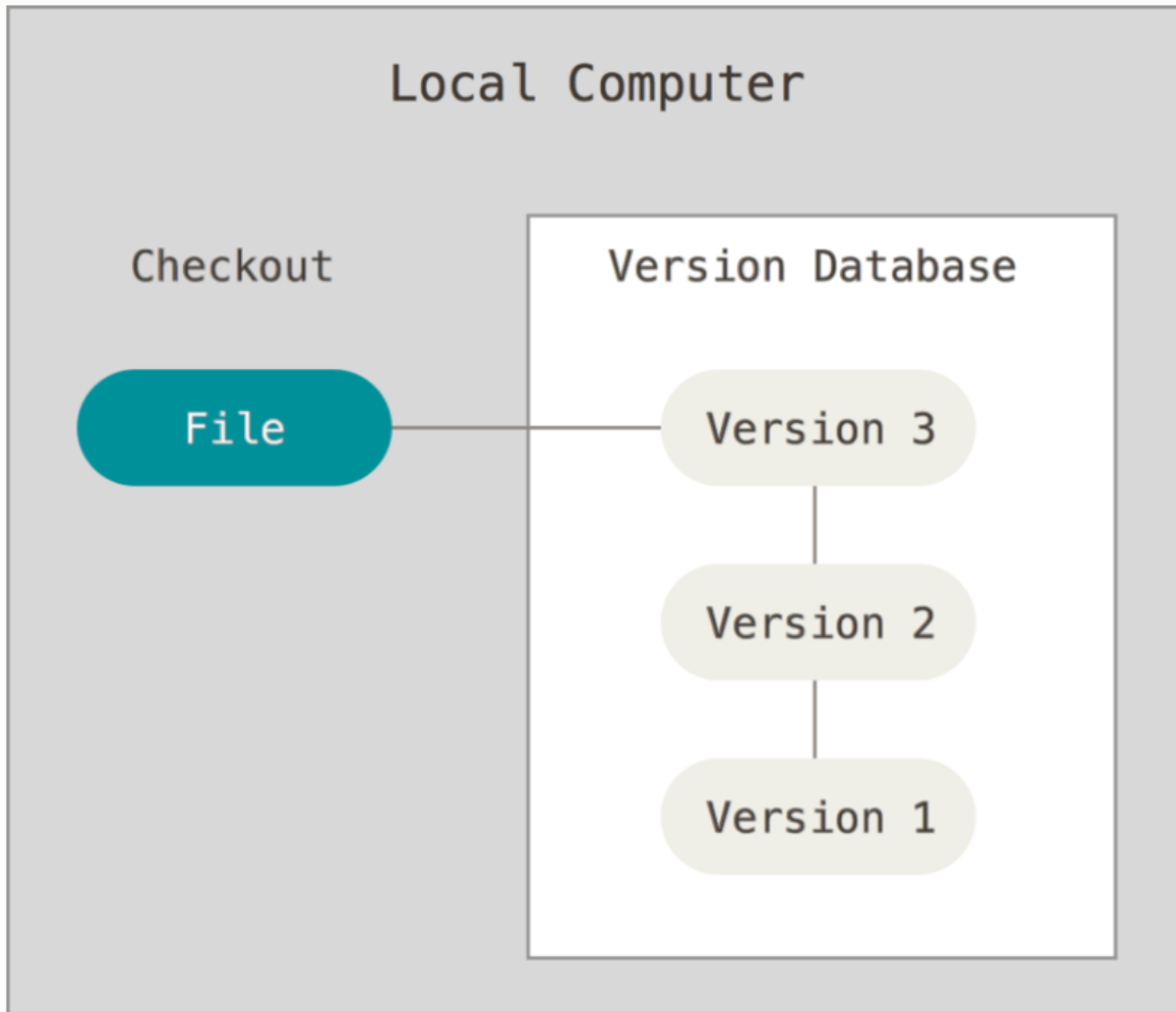


Simple, but incredibly ERROR PRONE

- You can forget in which directory you are in
- Write the wrong file
- Copy over files you do not mean to.

Then, programmers long ago developed **LOCAL VCSs** with a simple database, which kept all changes to files under revision control.

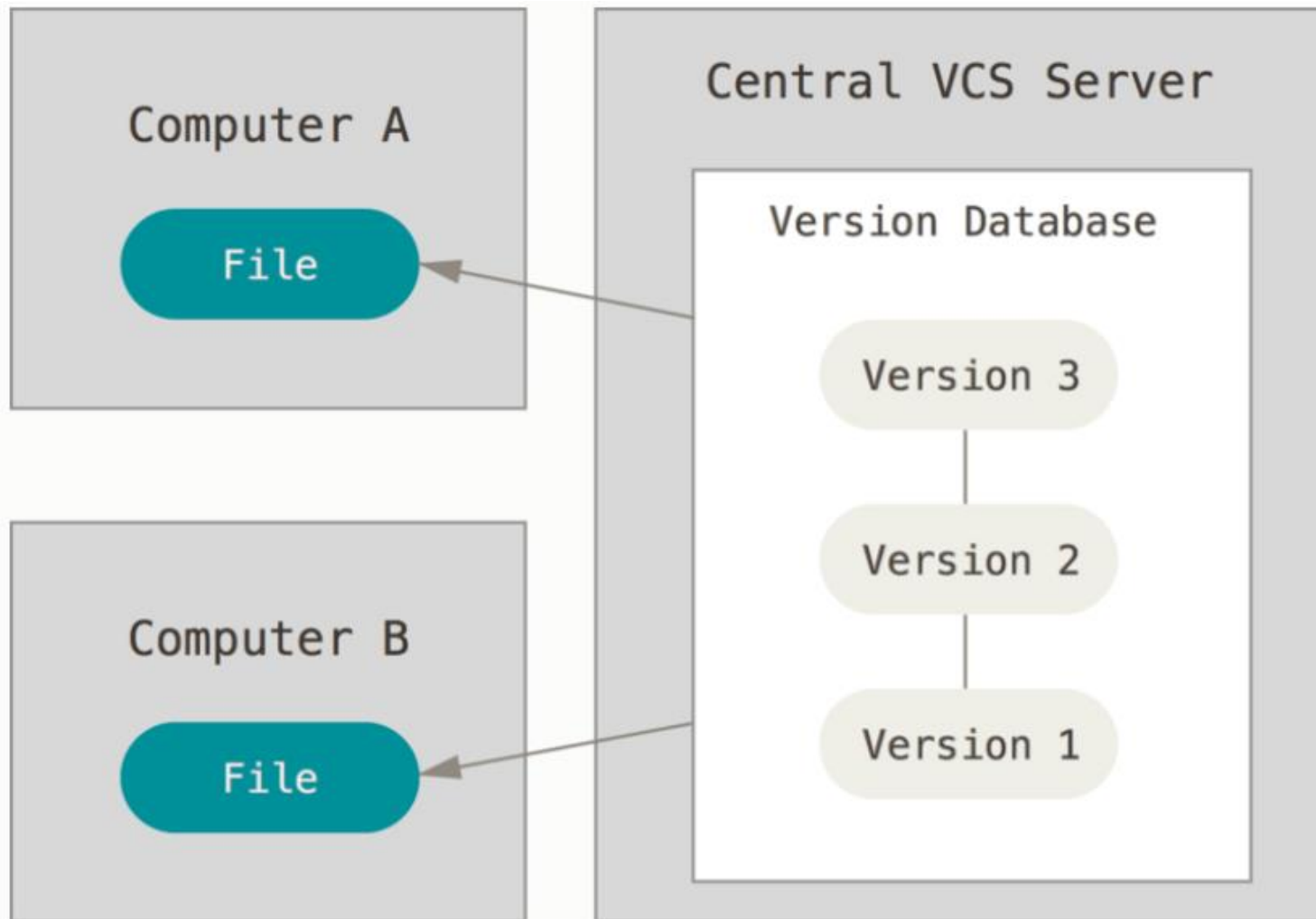
Local Version Control Systems Schema



The database locally stores all different versions of a file (Version 1, Version 2, Version 3 in the picture).

When checking out the file, a specific version can be accessed as needed.

Centralized Version Control Systems



What about people collaborating each other on different systems? **CVCSs**

These systems (such as CVS, Subversion, and Perforce) have a single server containing all the versioned files and a number of clients that check out files from that central place.

CVCSs PROs

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

CVCSs CONs

The centralized server represents a **single point of failure**:

- If the server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they are working on.
- If the hard disk the central database is on becomes corrupted, and proper backups have not been kept, everything will be absolutely lost.

NB

Local VCS systems suffer from this same problem.

Distributed Version Control Systems

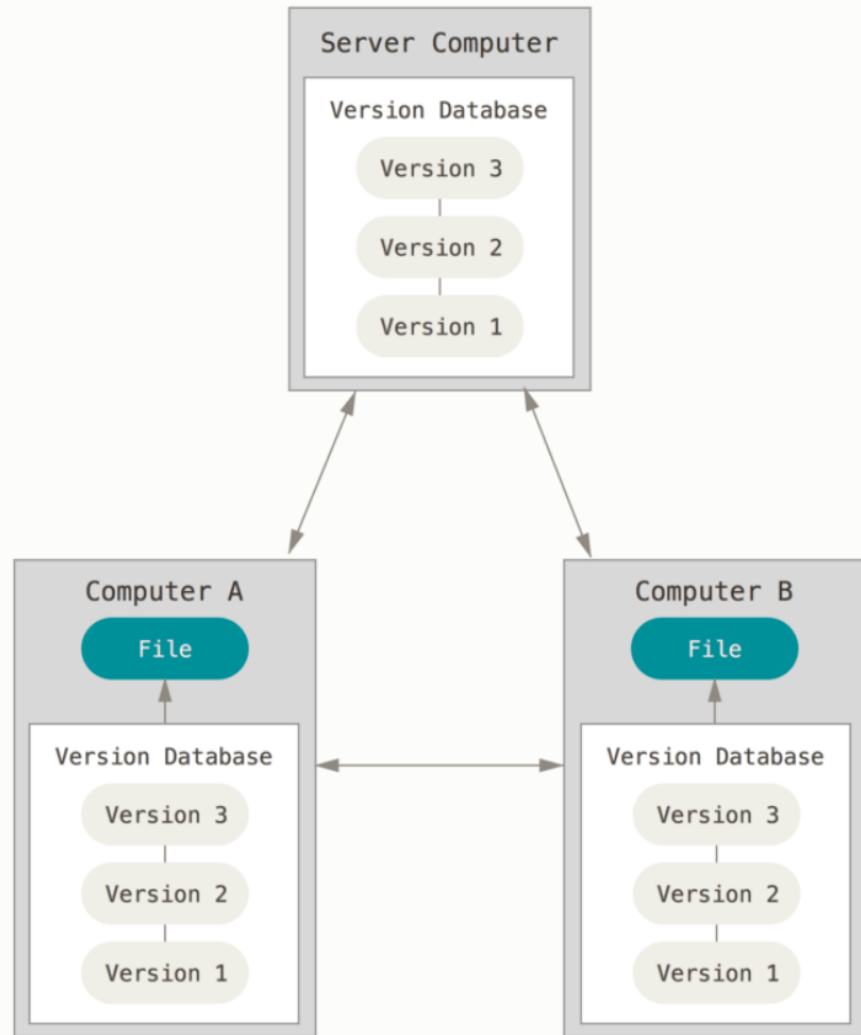
Distributed Version Control Systems (DVCSs) face to previously described issues.

In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients do not just check out the latest snapshot of the files: **they fully mirror the repository**, including its full history.

Thus, if any server dies and there were system collaborating via that server, any of the client repositories can be copied back up to the server to restore it.

Every checkout is really a full backup of all the data.

Distributed Version Control Systems schema



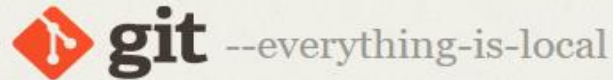
Many of these systems deal pretty well with having several remote repositories they can work with.

It is possible to collaborate with different groups of people in different ways simultaneously within the same project.

This allows to set up several types of workflows that are not possible in centralized systems.

GIT

What is Git?



Search entire site...

Git is a **free and open source** distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is **easy to learn** and has a **tiny footprint with lightning fast performance**. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, convenient **staging areas**, and **multiple workflows**.



About

The advantages of Git compared to other source control systems.



Documentation

Command reference pages, Pro Git book content, videos and other material.



Downloads

GUI clients and binary releases for all major platforms.



Community

Get involved! Bug reporting, mailing list, chat, development and more.



Pro Git by Scott Chacon and Ben Straub is available to **read online for free**. Dead tree versions are available on [Amazon.com](#).



Windows GUIs



Tarballs



Mac Build



Source Code

<https://git-scm.com/>

Git is free and open source

Git is released under the GPLv2 open source license, that means you are free to inspect the source code at any time or contribute to the project yourself.

You may:

- use Git on open or proprietary projects for free, forever
- download, inspect and modify the source code to Git
- make proprietary changes to Git that you do not redistribute publicly
- call Git binaries from your open or proprietary programs
- publicly redistribute Git binaries with open or proprietary programs, given that they are unmodified or the modifications are public.

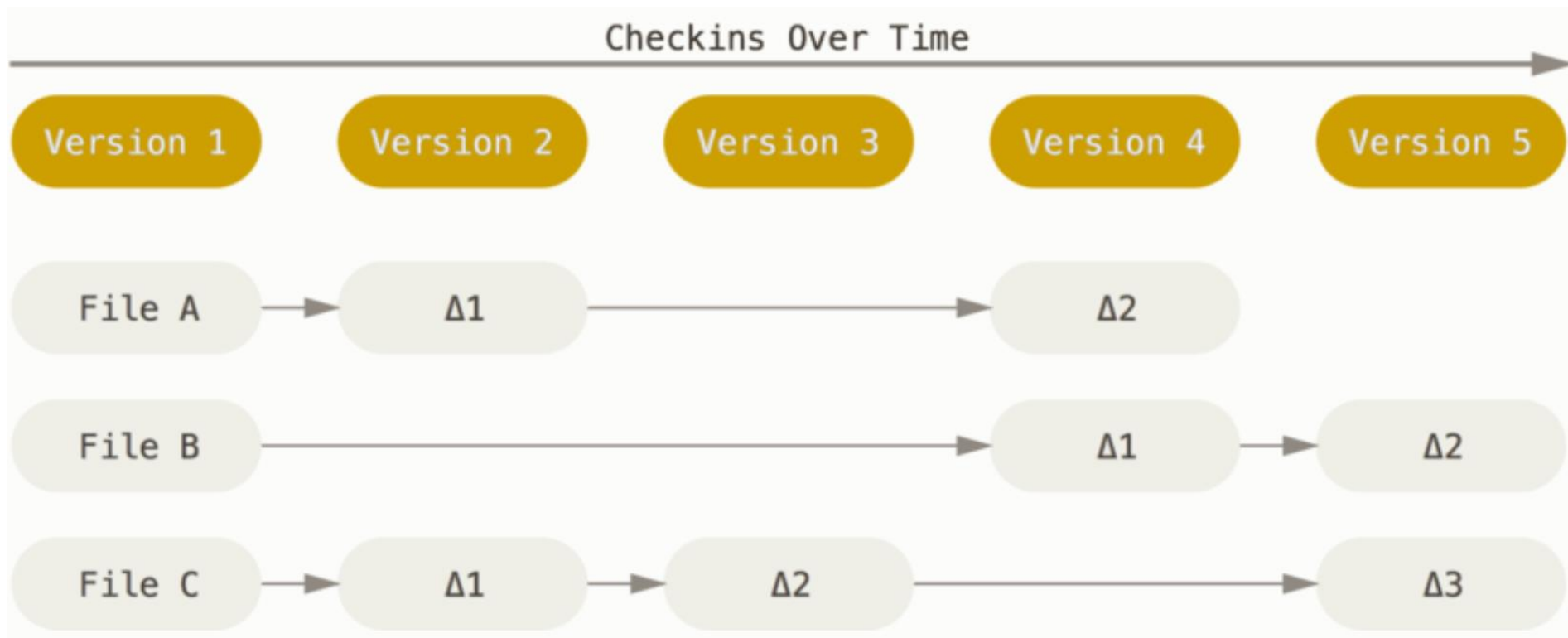


You may not:

- make proprietary changes to Git and publicly redistribute it without sharing the changes
- make and publicly distribute changes to Git under a different license
- use source code from the Git repository in a project under a different license without permission

Any VCS way to think data

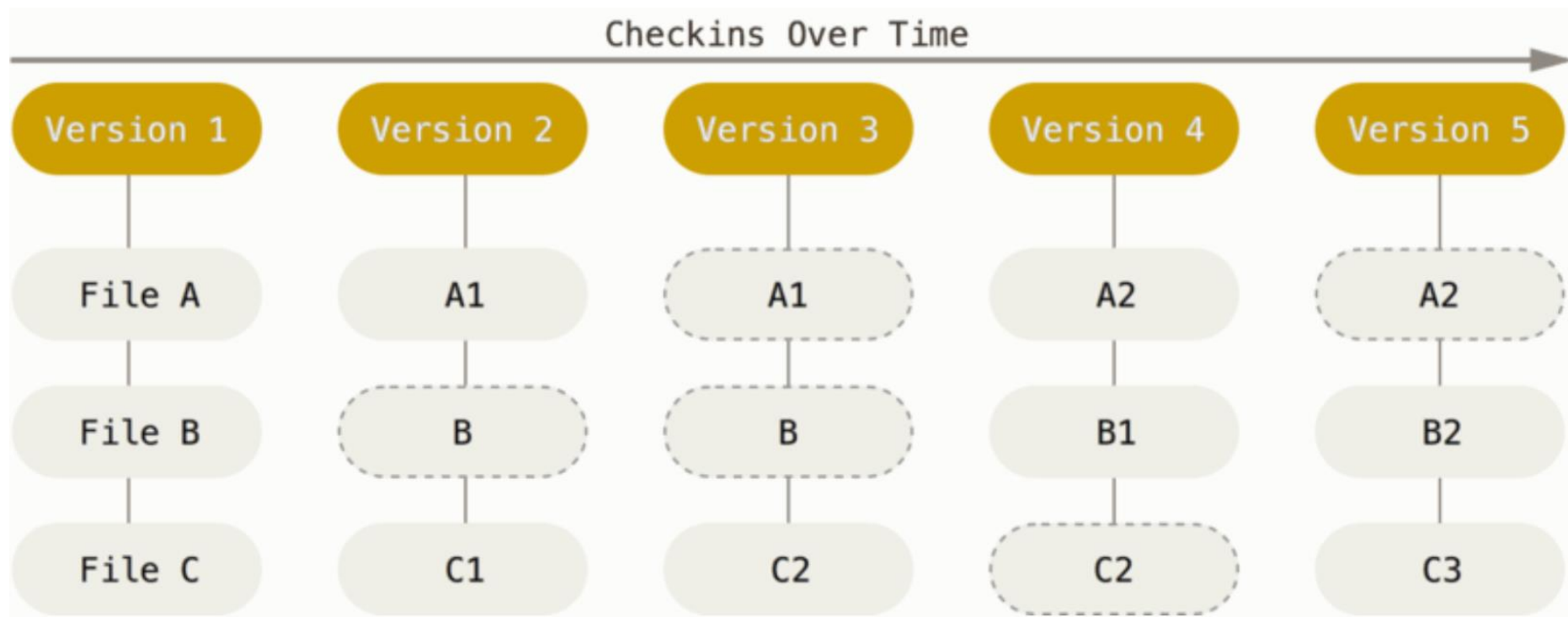
The major difference between Git and any other VCS is how to think about data. Systems as CVS, Subversion, Perforce, Bazaar and so on, store the information as a list of file-based changes (delta-based version control).



Git way to think data

Git thinks of its data more like **a series of snapshots** of a miniature filesystem.

When required, Git basically takes a picture of what all files look like at that moment and stores a reference to that snapshot. If files have not changed, the file is not stored again, there is just a link to the previous identical file it has already stored.



Git, nearly every operation is Local

Most operations in Git only need local files and resources to operate (generally no information is needed from another computer on your network), that is why they seem almost instantaneous.

Git allows to see changes between the current version of a file and a previous one, just doing a local difference calculation.

IE

Git looks up the file of a month ago and does a local difference calculation, instead of having to either ask a remote server to do it, either pull an older version of the file from the remote server.

There is very little you cannot do if you are offline!

Git has Integrity

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it is impossible to change the contents of any file or directory without Git knowing about it and it is not possible to lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this check-summing is called a SHA-1 hash, a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.

Ex. `24b9da6552252987aa493b52f8696cd6d3b00373`

These hash values are all over the place in Git, it stores everything in its database not by file name but by the hash value of its contents.

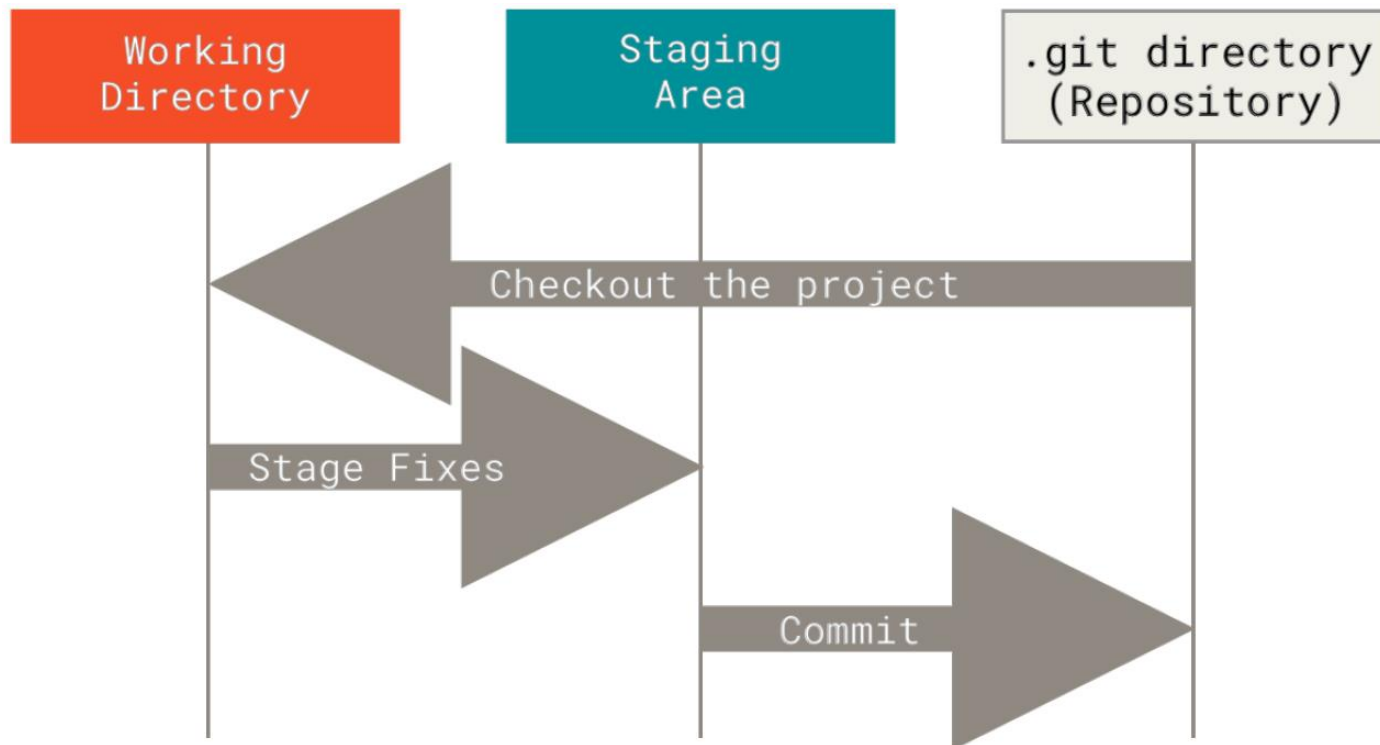
Git has Three States

Git has three main states that files can reside in: **committed**, **modified**, and **staged**.

- Committed means that the data is safely stored in the local database.
- Modified means that the file has been changed but not yet committed to the database.
- Staged means that a modified file has been marked in its current version to go into the next commit snapshot.

This leads us to the three main sections of a Git project: the **git directory**, the **working directory** and the **staging area**.

Git, three states schema



The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk to be used or modified.

The staging area is a file, generally in the Git directory, that stores information about what will go into the next commit.

The Git directory is where Git stores the metadata and object database for the project. This is the most important part of Git and it is what is copied when **cloning** a repository from another computer.

The basic Git workflow

The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You selectively stage files, just those you want to be part of your next commit, which adds snapshots of them to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Some basic terms

- **Clone:** if you want to get a copy of an existing Git repository — for example, a project you would like to contribute to — the command you need is `Git clone`.
- **Checkout:** the working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- **Tracked vs. Untracked:** 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. Untracked files are everything else.

Some basic terms

- **Push and Pull:** collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.
- **Branching:** branching means you diverge from the main line of development and continue to do work without messing with that main line. The way Git branches is incredibly lightweight, making branching operations nearly instantaneous and switching back and forth between branches generally just as fast.
- **Merging:** to merge branches

Branching and Merging

Git allows and encourages you to have multiple local branches that can be entirely independent of each other.

Things can be done like:

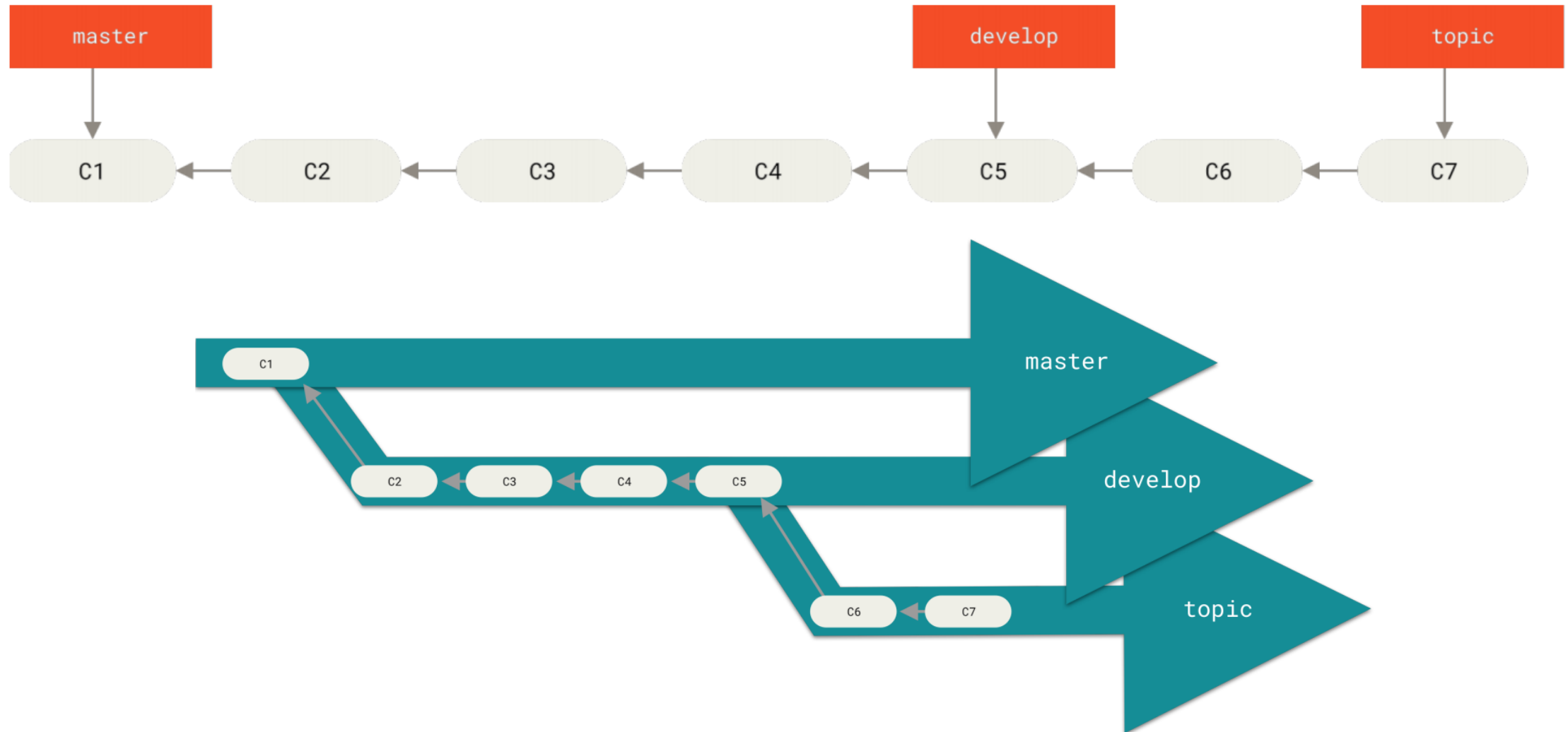
- **Frictionless Context Switching.** Create a branch to try out an idea, commit a few times, switch back to where you branched from, apply a patch, switch back to where you are experimenting and merge it in.
- **Role-Based Codelines.** Have a branch that always contains only what goes to production, another that you merge work into for testing and several smaller ones for day to day work.

Branching and Merging

(Things can be done like:)

- **Feature Based Workflow.** Create new branches for each new feature you are working on so you can seamlessly switch back and forth between them, then delete each branch when that feature gets merged into your main line.
- **Disposable Experimentation.** Create a branch to experiment in, realize it is not going to work, and just delete it - abandoning the work - with nobody else ever seeing it (even if you have pushed other branches in the meantime).

Branching and Merging



Branching and Merging

In master branch there is only code that is entirely stable, possibly only code that has been or will be released.

In another parallel branch named develop or next, there is code to work from or use to test stability; it is not necessarily always stable, but whenever it gets to a stable state, it can be merged into master.

A topic branch is a short-lived branch, created and used for a single particular feature or related work.

The idea: branches are at various levels of stability; when they reach a more stable level, they are merged into the branch above them.

Some Git resources

- <http://git-scm.com/> (Git site)
- <http://git-scm.com/book/en> (Book)
- <http://git-scm.com/downloads> (the Git software)
- <http://code.google.com/p/gitextensions/> (a Git GUI)
- <http://www.eclipse.org/egit/>
(EGit is an Eclipse Team provider for the Git version control system)

Git Commands

git config

git config

- **Set your identity**

Set your user name and e-mail address (git commit uses this information).

```
$ git config --global user.name "Name Surname"
```

```
$ git config --global user.email namesurname@example.com
```

Operation needed just once. The --global option allows git to use the information for anything you do on that system.

If you want to override this with a different name or email address for specific projects, you can run the command without the --global option directly in that project.

git config

git config

- **Check your settings**

To check the configurations settings and see a list them all

```
$ git config --list
```

- **check what Git knows about a specific key's value**

```
$ git config <key>
```

Ex.

```
$ git config user.name
```

Result in the git bash window

```
user.name=Name Surname  
user.email=namesurname@example.com  
color.status=auto  
color.branch=auto  
color.interactive=auto  
color.diff=auto
```

```
Name Username
```

git help

git help

Three different ways to get help:

```
$ git help <verb>
```

```
$ git <verb> --help
```

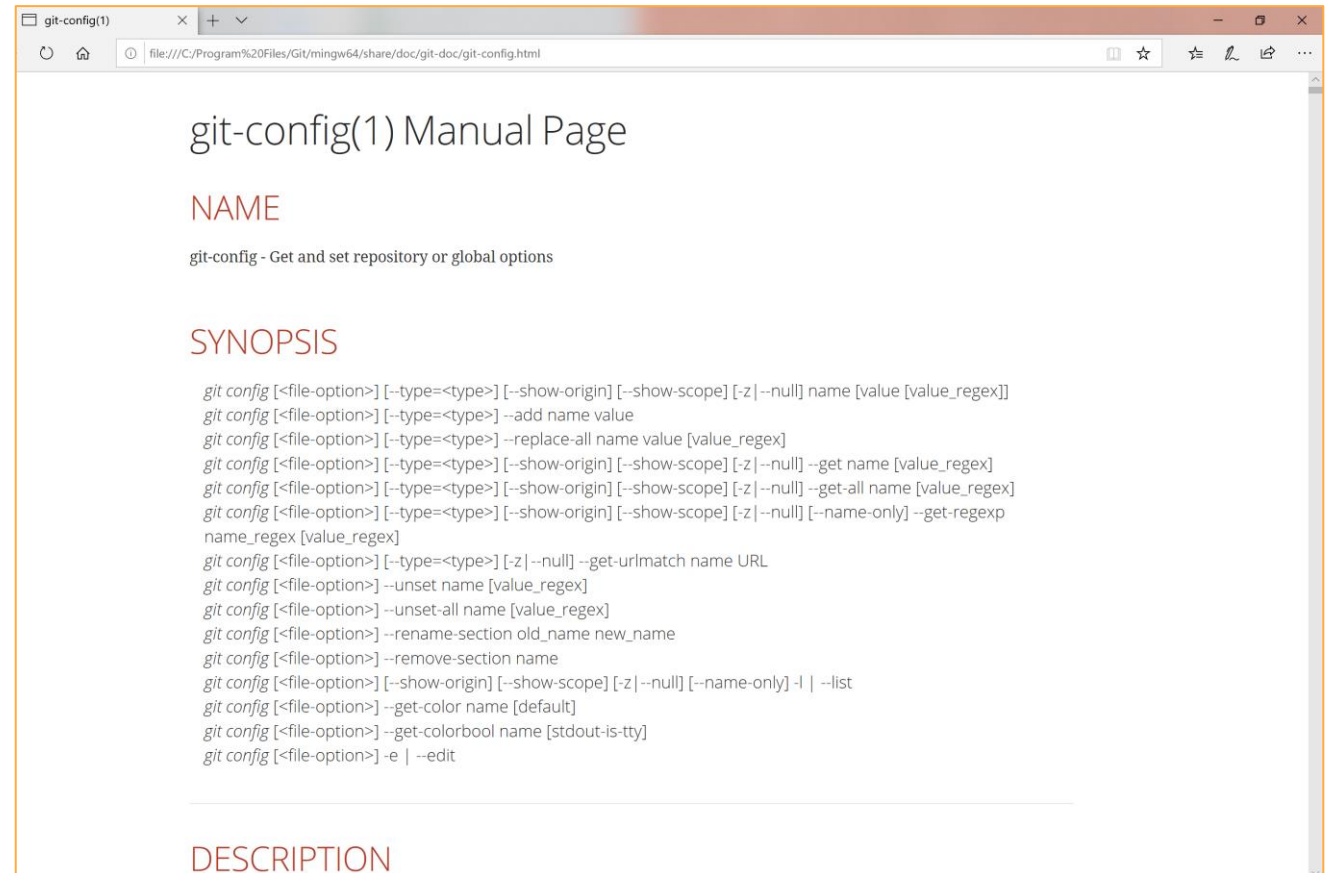
```
$ man git-<verb>
```

Ex.

```
$ git help config
```

→ A local page will show up

<file:///C:/Program%20Files/Git/mingw64/share/doc/git-doc/git-config.html> (example on Windows)



How to get a git repository

Getting a Git Repository

A Git repository can be obtained in two ways:

- Take a local directory that is currently not under version control and turn it into a Git repository
- **Clone** an existing Git repository from elsewhere.

In either case, there will be a Git repository on the local machine, ready for work.

Initialize a repository

In the git bash window:

1. Go to the project's directory.

Linux

```
$ cd /home/user/my_project
```

MacOS

```
$ cd /Users/user/my_project
```

Windows

```
$ cd C:/Users/user/my_project
```

2. Initialize

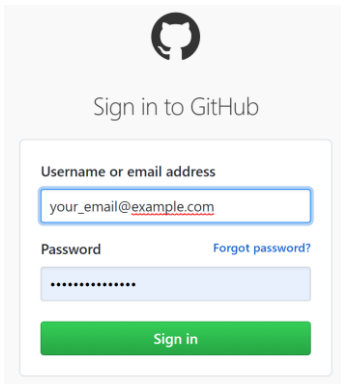
3. Add files

4. Commit

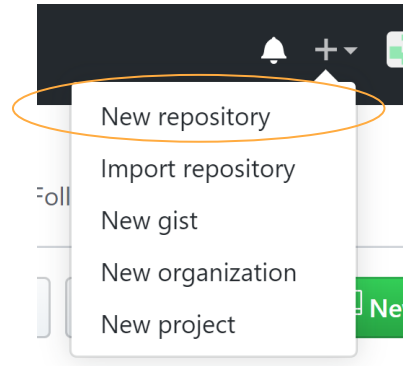
```
$ git init  
$ git add .  
$ git commit -m 'Initial project version'
```

Initialize a repository: upload on github

1.



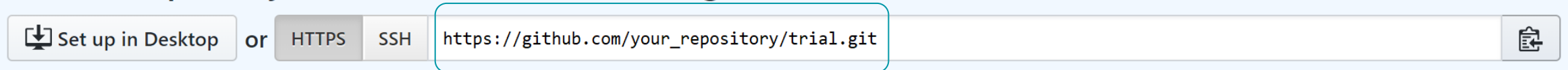
2.



1. Log in on your github account
2. Create a new repository
3. Select remote url
4. Type in your git bash following commands (only where there is a \$)

3.

Quick setup — if you've done this kind of thing before



4.

```
$ git remote add origin https://github.com/your_repository/trial.git
# Sets the new remote
$ git remote -v
# Verifies the new remote URL
$ git push origin master
# Pushes the changes in your local repository up to the remote repository you
specified as the origin
```

Cloning an existing repository

Every version of every file for the history of the project is pulled down by default when you run git clone.

```
$ git clone <url>
```

Ex.

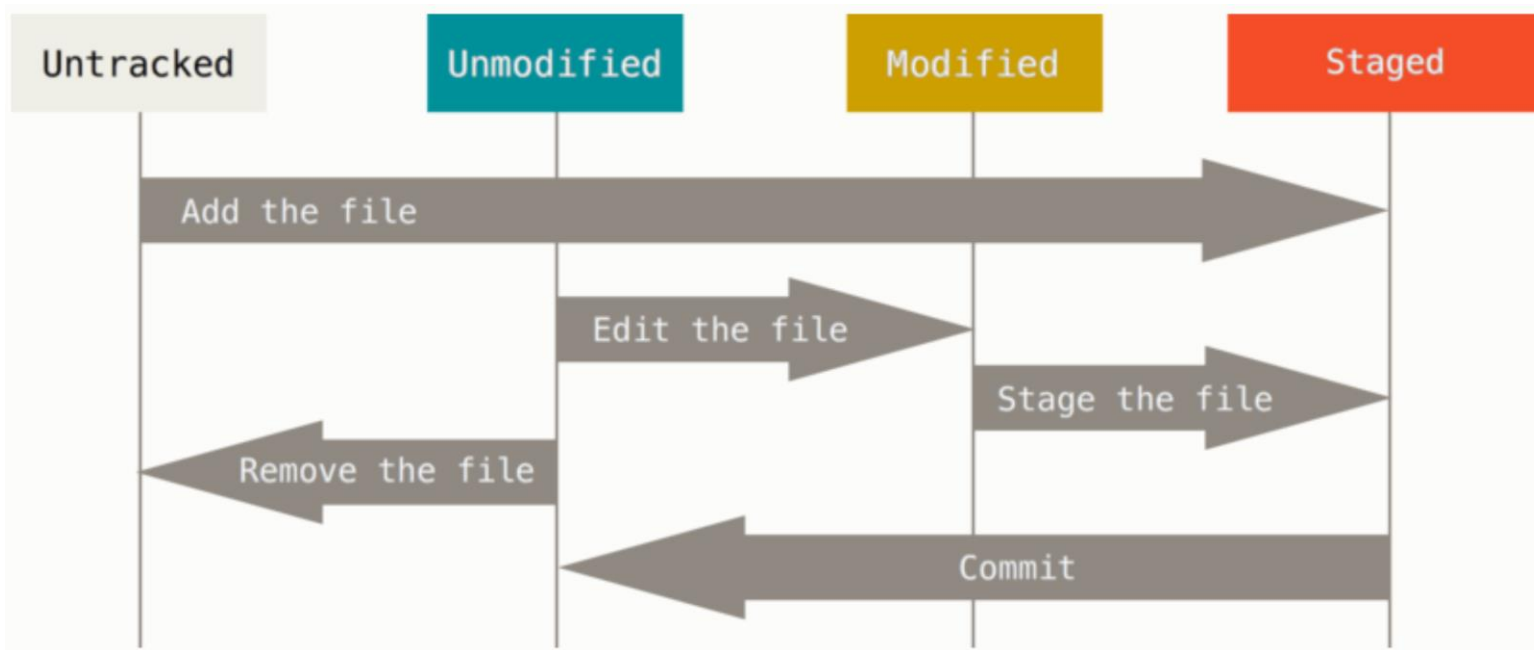
```
$ git clone https://github.com/octocat/Hello-World.git
```

This creates a directory named Hello-world, initializes a .git directory inside it, pulls down all the data for that repository and checks out the latest version.

To clone the repository into a directory named something other than Hello-world.

```
$ git clone https://github.com/octocat/Hello-World.git myHelloWorld
```

The lifecycle of the status of your files



```
$ git add <files>
```

Tracking new files, stage files, marking merge-conflicted files as resolved

```
$ git diff
```

To see what you've changed but not yet staged

```
$ git diff -staged
```

To compares the staged changes with the last commit

```
$ git status
```

```
$ git commit
```

```
$ git rm
```

Removes a file from the staging area and delete it

git status

To check in which state is a file

```
$ git status
```

What does it happen when running git status after a clone?

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

What does it happen when running git status after creating a new file?

```
$ echo 'My Project' > CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  CONTRIBUTING.md
nothing added to commit but untracked files present (use "git add" to track)
```

git add

To begin tracking a new file

```
$ git add <file>
```

What does it happen when running git status after adding a file?

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   CONTRIBUTING.md
```

git commit

When the staging area is set up properly, it is possible to commit changes:
\$ git commit

Anything that is still unstaged (any file created or modified that has not been added by using git add, since it has been edited) will not go into this commit.

It is a good practice to add a message to comment on the changes made with the last commit.

```
$ git commit -m "Story 182: fix benchmarks for speed"
```


Staging modified files

Let's change a file that was already tracked. After modifying the file and running `git status`:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

To stage it, run the `git add` command.

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md
```

Staging modified files

What does it happen if after adding a file (`git add <file>`) you make some changes?

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

How can it be possible to have the same file in two different phases?
Which one is the right one?

git diff

To know exactly what has changed inside the files, not just which files were changed:

```
$ git diff
```

git diff shows the exact lines added and removed.

The git diff command exactly allows to verify what has changed but not yet staged.

To get what has changed and it is already staged:

```
$ git diff --staged      or      $ git diff --cached
```

Viewing your staged and unstaged changes

Let's say you edit and stage the README file and then edit the CONTRIBUTING.md file without staging it.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

N.B. README is staged, ready to be committed
CONTRIBUTING.md is unstaged, not ready to be committed.

Viewing your staged and unstaged changes

Running git diff

```
$ git diff
warning: LF will be replaced by CRLF in
CONTRIBUTING.md.
The file will have its original line
endings in your working directory
diff --git a/CONTRIBUTING.md
b/CONTRIBUTING.md
index af0abd8..d3ef71d 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -1,1 +1,2 @@
-Hola, estoy aqui
+Hola, estoy aqui. Y tu?
```

Running git diff --staged

```
$ git diff --staged
diff --git a/README b/README
index 2726da5..7dffd7b 100644
--- a/README
+++ b/README
@@ -1,1 +1,2 @@
-Prova.
+Prova. Riprova.
```

Viewing your staged and unstaged changes

Let's now stage the CONTRIBUTING.md file and then modify it.

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

Which one is the staged file? `$ git diff --staged`

Which one is the unstaged file? `$ git diff`

git rm

To remove a file from Git, it is necessary to remove it from tracked files (more accurately, remove it from staging area) and then commit.

```
$ git rm <file>
```

Another useful thing is to keep the file in the local working tree, but remove it from the staged files on the original repository.

```
$ git rm --cached <file>
```

git log

To see what has happened in a project:

\$ git log

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

Change version number

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

Remove unnecessary test

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

Initial commit

Undoing things

Already committed and forgot to add some files:

```
$ git commit -m 'Initial commit'  
$ git add forgotten_file  
$ git commit --amend
```

Changed two files and want to commit them as two separate changes, but you accidentally type `git add *` and stage them both.

```
$ git add *  
$ git reset HEAD CONTRIBUTING.md
```

You changed the content of a file, it is not staged and you do not want to keep it, but you want to go back to the last committed version

```
$ git checkout -- CONTRIBUTING.md
```

N.B. Undoing things is discouraged, pay attention during the regular lifecycle.

git branching

Git branching? → THE KILLER FEATURE

Why is it so special?

The way Git branches is incredibly lightweight.

Branching operations are nearly instantaneous and switching back and forth between branches generally just as fast

Git encourages a workflow that branches and merges often, even multiple times in a day.

git branching: how files are stored

Git stores data as a series of **snapshots**

When a commit is made, Git stores a commit object that contains a pointer to the snapshot of the content staged.

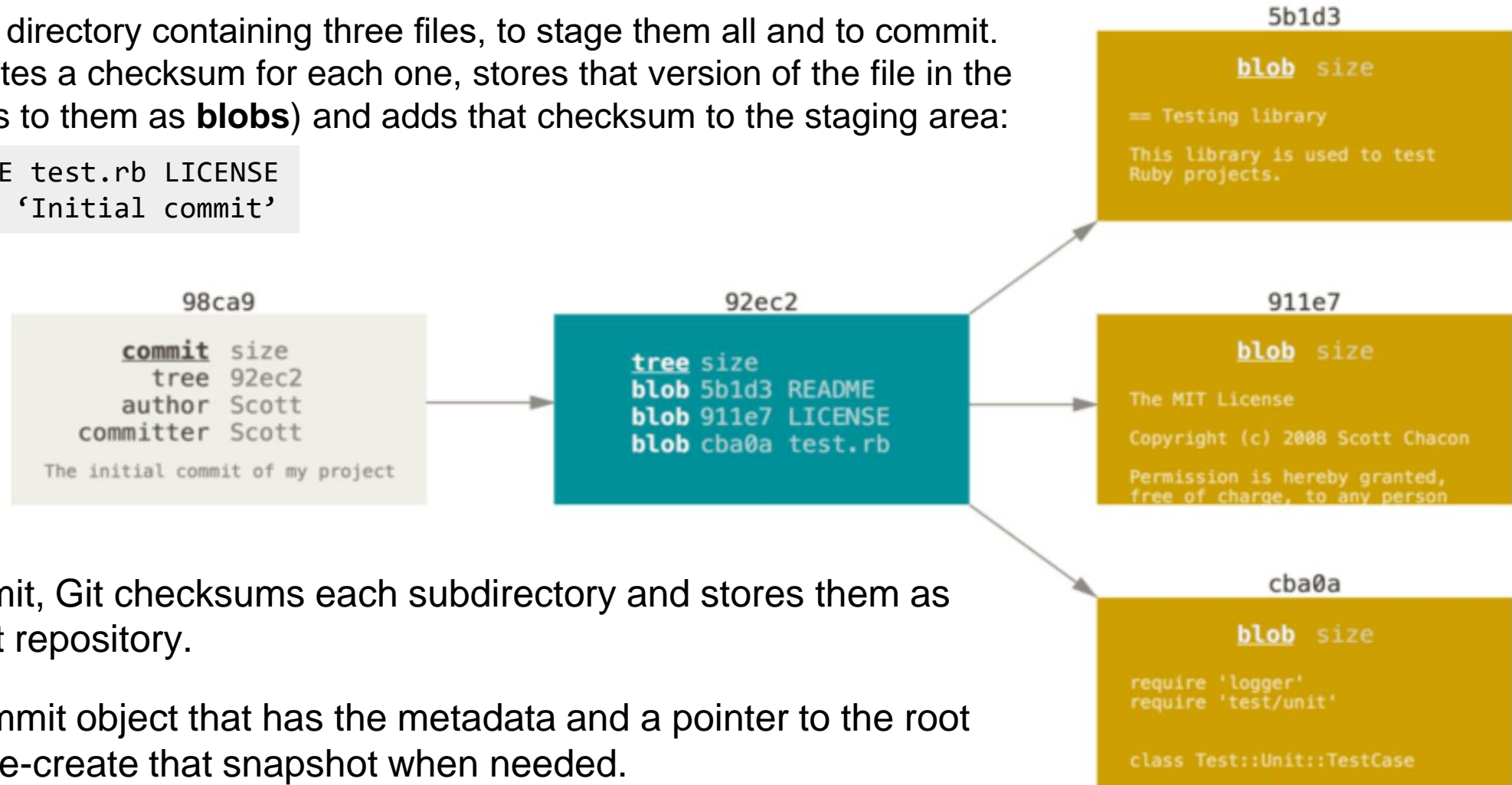
This object also contains:

- author's name and email address
- the message typed in the commit
- pointers to the commit or commits that directly came before this commit:
zero parents for the initial commit, one parent for a normal commit and multiple parents for a commit that results from a merge of two or more branches.

git branching: how files are stored

Let's assume to have a directory containing three files, to stage them all and to commit. Staging the files computes a checksum for each one, 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 'Initial commit'
```

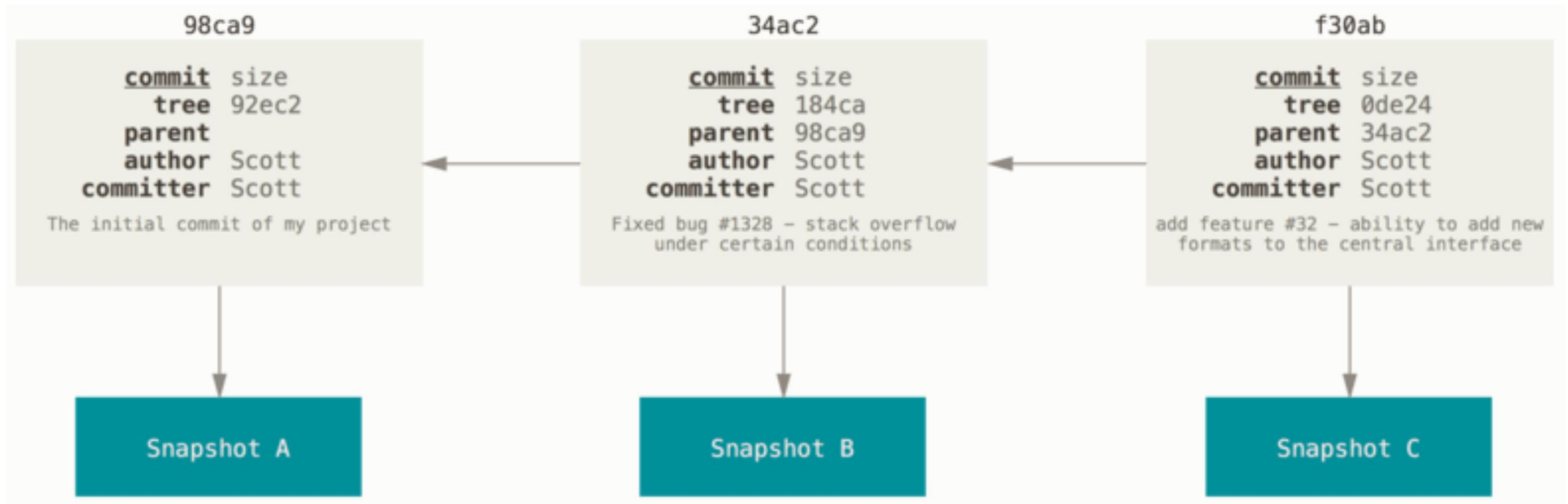


After running git commit, Git checksums each subdirectory and stores them as a tree object in the Git repository.

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.

git branching: how files are stored

What does it happen when making some changes and committing again?
The next commit stores a pointer to the commit that came immediately before it.

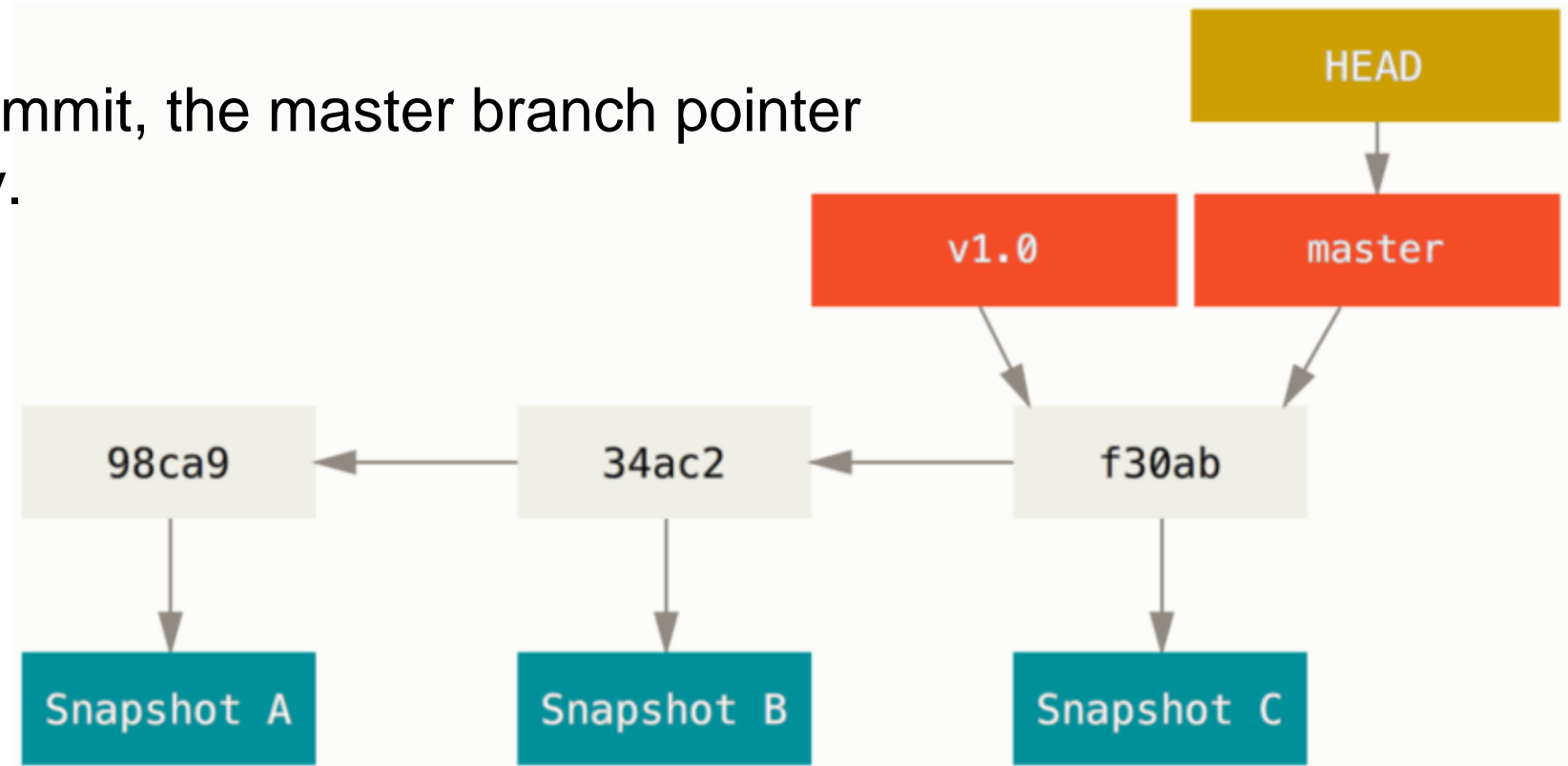


git branching: how files are stored

A branch is a lightweight movable pointer to one of the commits.
The default branch name is master, that, after starting making commits, points to the last commit made.

Every time there is a new commit, the master branch pointer moves forward automatically.

The “master” branch is not a special branch. It is exactly like any other branch.
The only reason nearly every repository has one is that the git init command creates it by default and usually nobody modifies it.



git branching: create a new branch

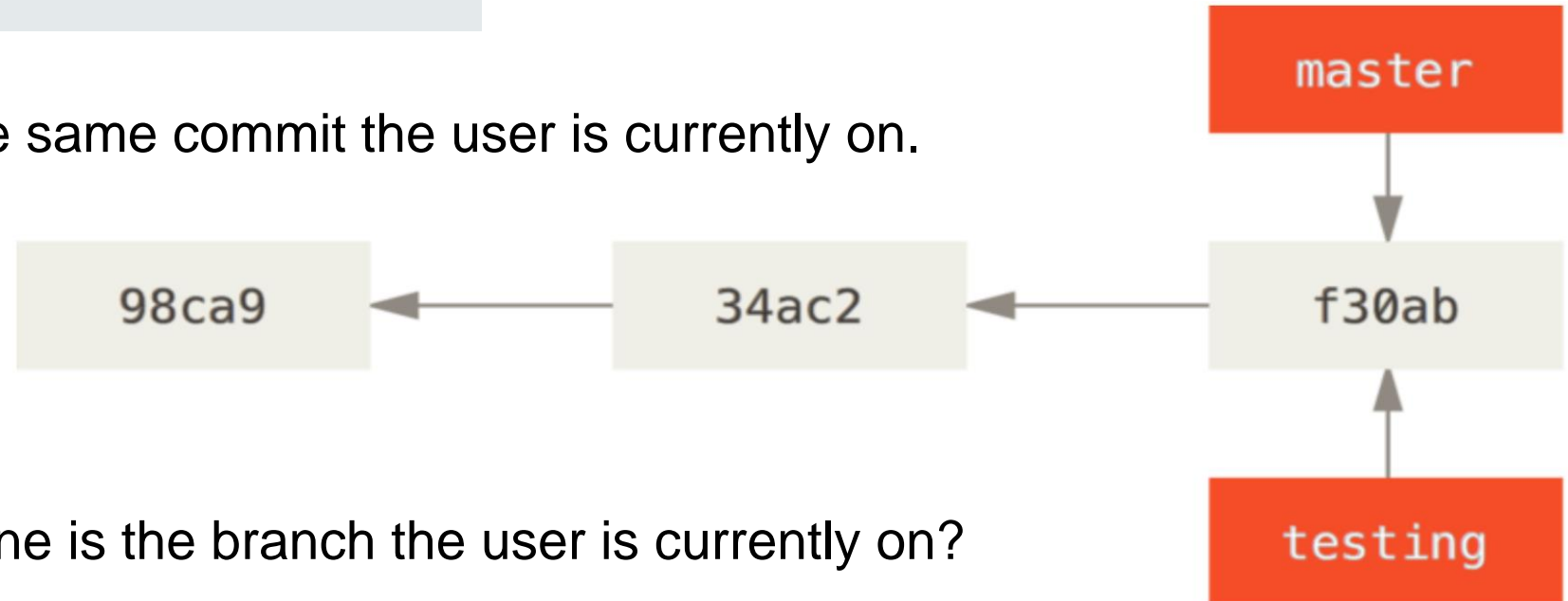
To add a new branch

```
$ git branch <new_branch>
```

Let's create a new branch, testing:

```
$ git branch testing
```

There will be a pointer to the same commit the user is currently on.




How does Git know which one is the branch the user is currently on?

git branching: create a new branch

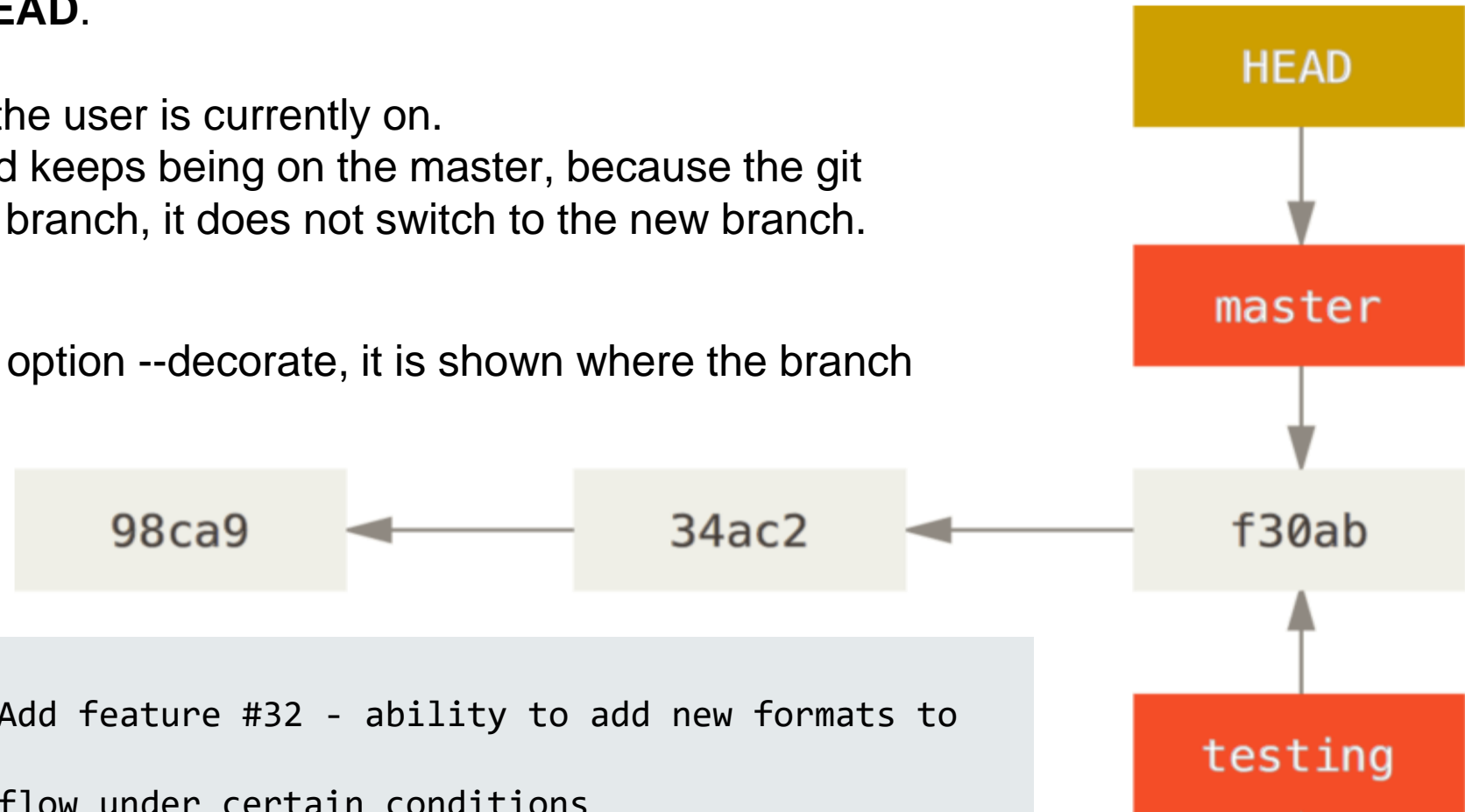
Git keeps a special pointer called **HEAD**.

This is a pointer to the local branch the user is currently on.
After creating a new branch the head keeps being on the master, because the git branch command only create a new branch, it does not switch to the new branch.

Running a git log command with the option --decorate, it is shown where the branch pointers are pointing.



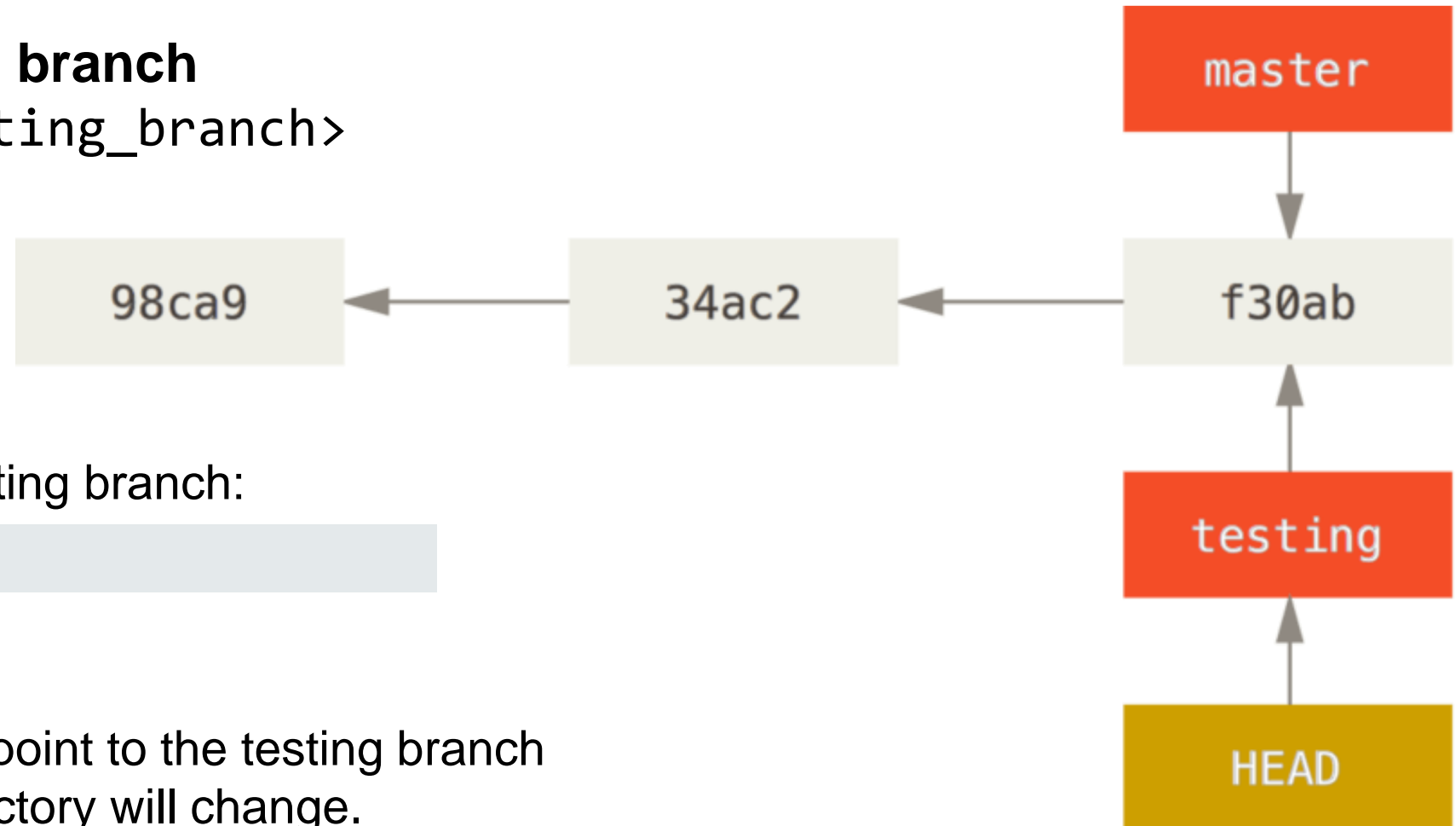
```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to
the central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```



git branching: switching branches

To switch to an existing branch

```
$ git checkout <existing_branch>
```



Let's switch to the new testing branch:

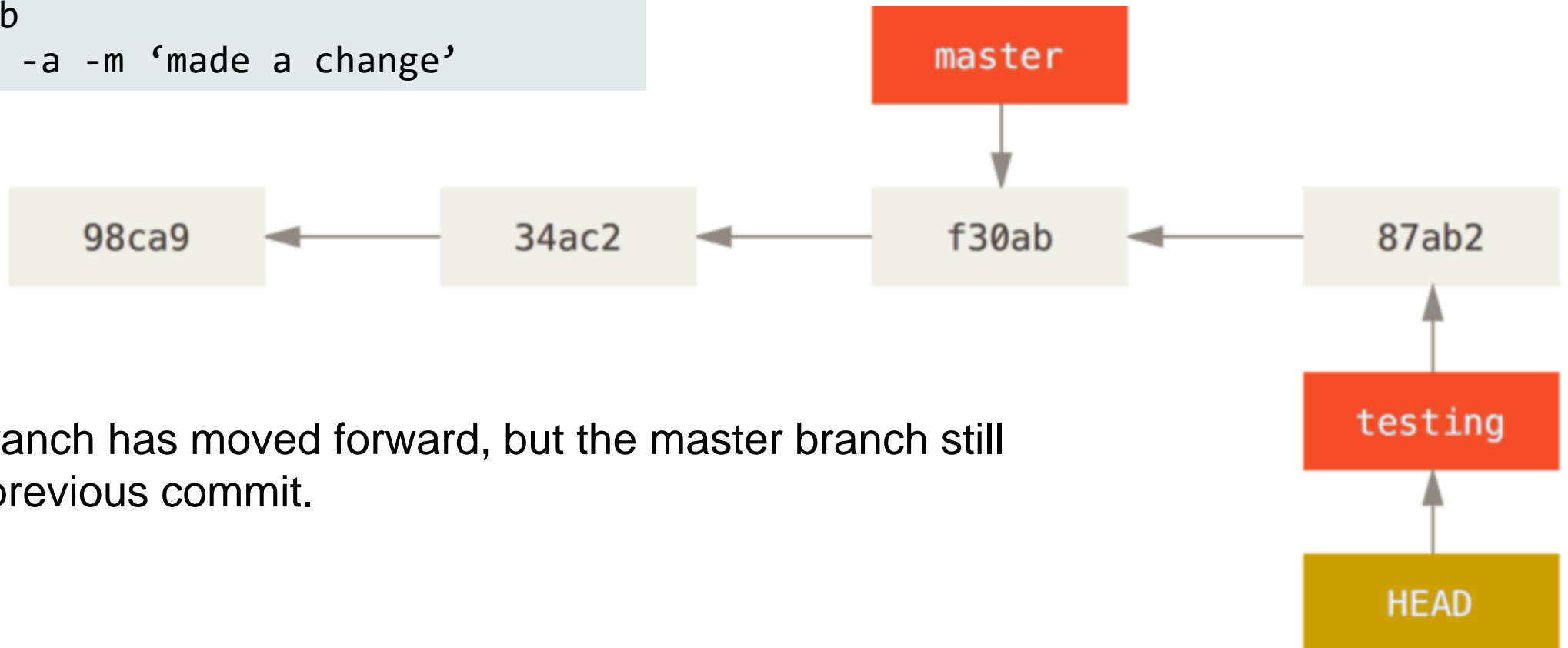
```
$ git checkout testing
```

- The HEAD is moved to point to the testing branch
- Files in the working directory will change.

git branching: switching branches

Let's make some changes and see what happens.

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



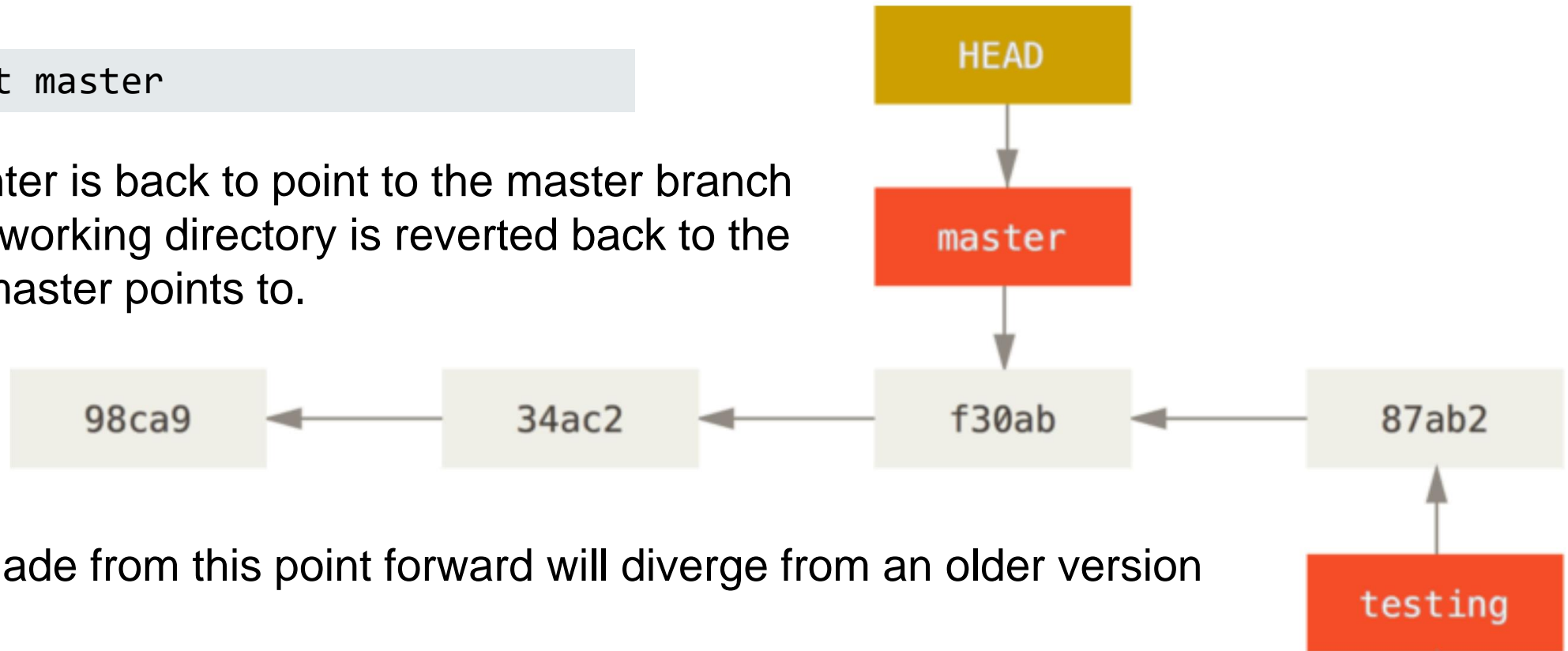
The testing branch has moved forward, but the master branch still points to the previous commit.

git branching: switching branches

Let's switch back to the master branch:

```
$ git checkout master
```

The HEAD pointer is back to point to the master branch
The files in the working directory is reverted back to the snapshot that master points to.



The changes made from this point forward will diverge from an older version of the project.

It essentially rewinds the work done in the testing branch so it is possible to go in a different direction.

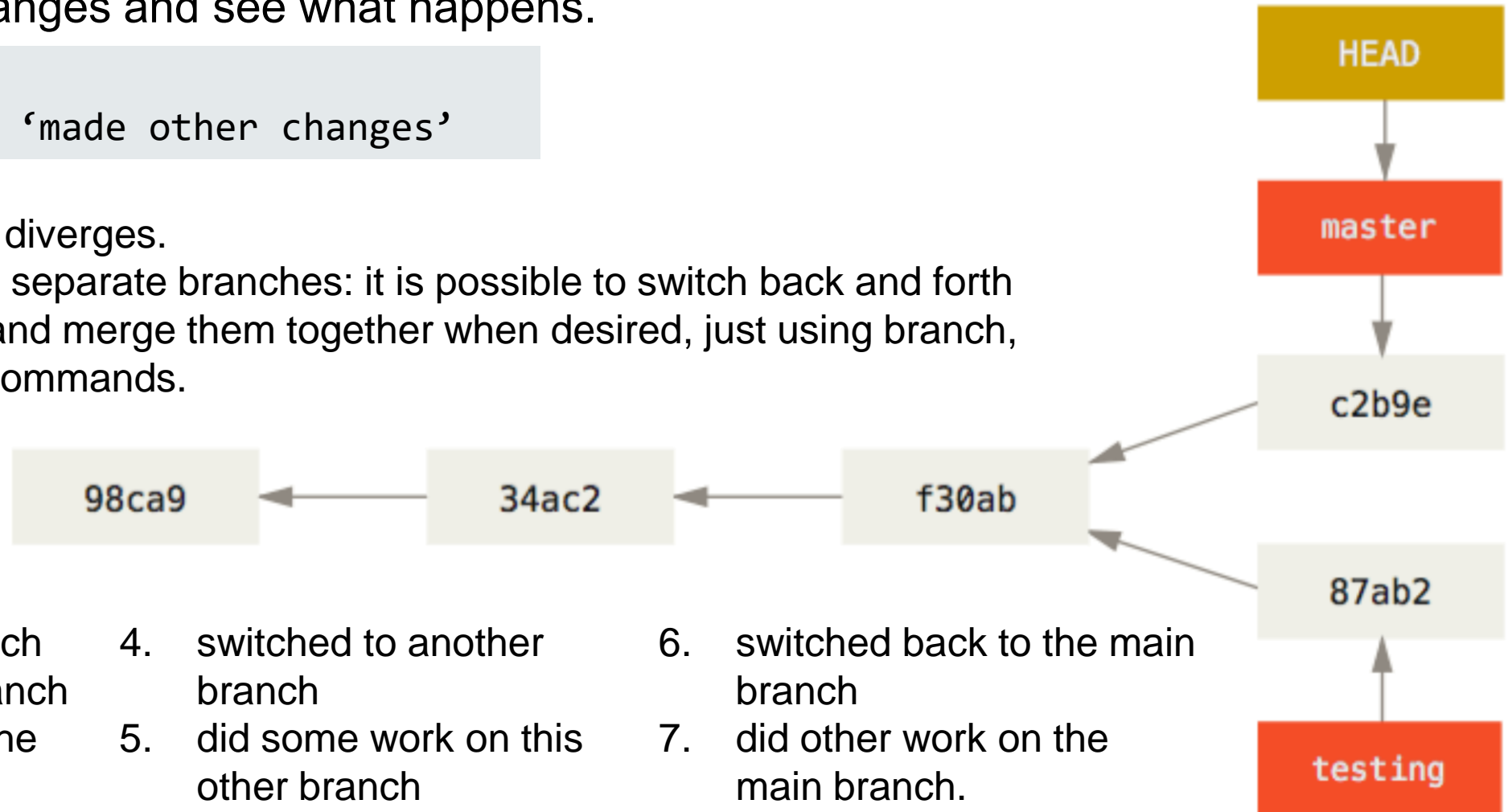
git branching: switching branches

Let's make some changes and see what happens.

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

The project history now diverges.

Changes are isolated in separate branches: it is possible to switch back and forth between the branches and merge them together when desired, just using branch, checkout, and commit commands.



We have:

- | | | |
|--------------------------------|---------------------------------------|---------------------------------------|
| 1. created a new branch | 4. switched to another branch | 6. switched back to the main branch |
| 2. switched to that branch | 5. did some work on this other branch | 7. did other work on the main branch. |
| 3. did some work on the branch | | |

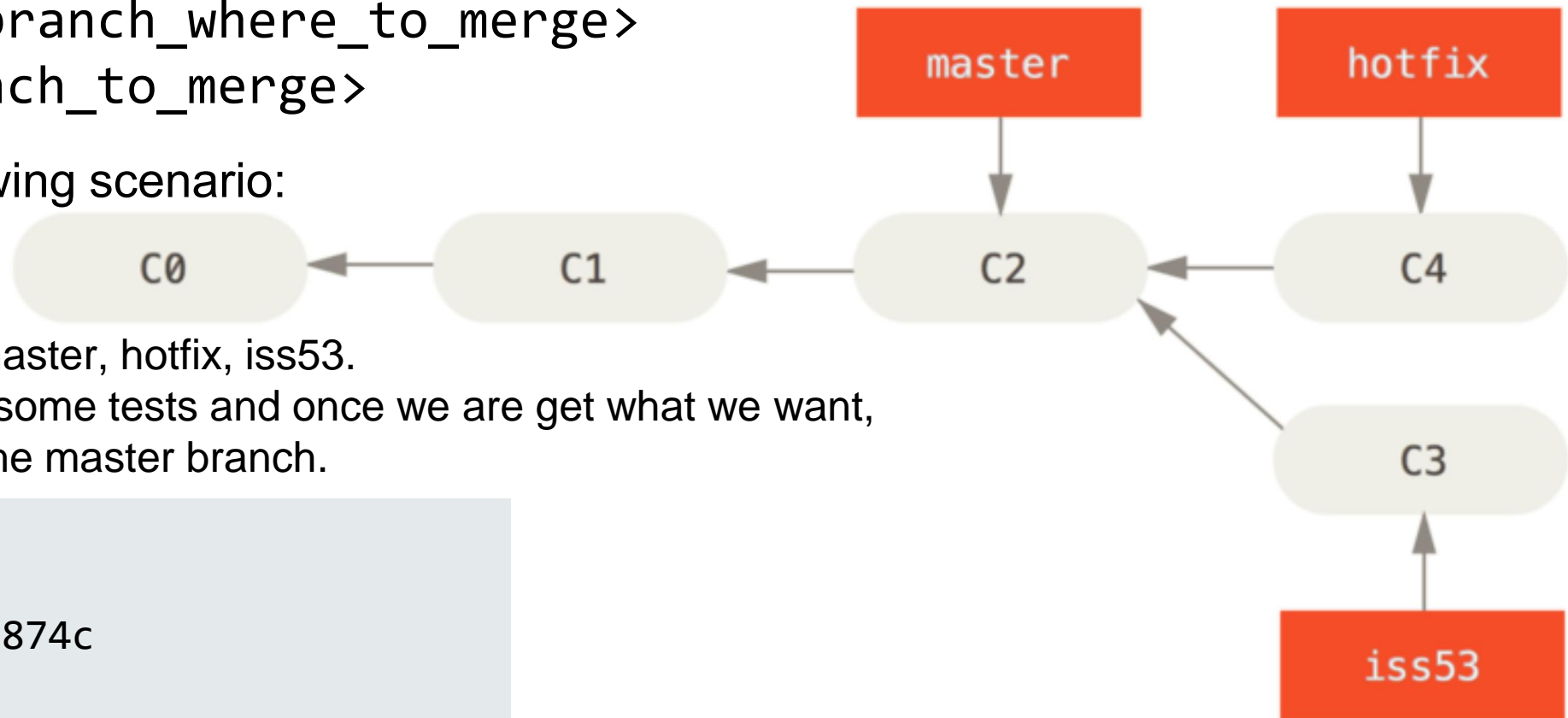
git merging

To merge two branches

```
$ git checkout <branch_where_to_merge>
```

```
$ git merge <branch_to_merge>
```

Let's consider the following scenario:



Three different branches: master, hotfix, iss53.

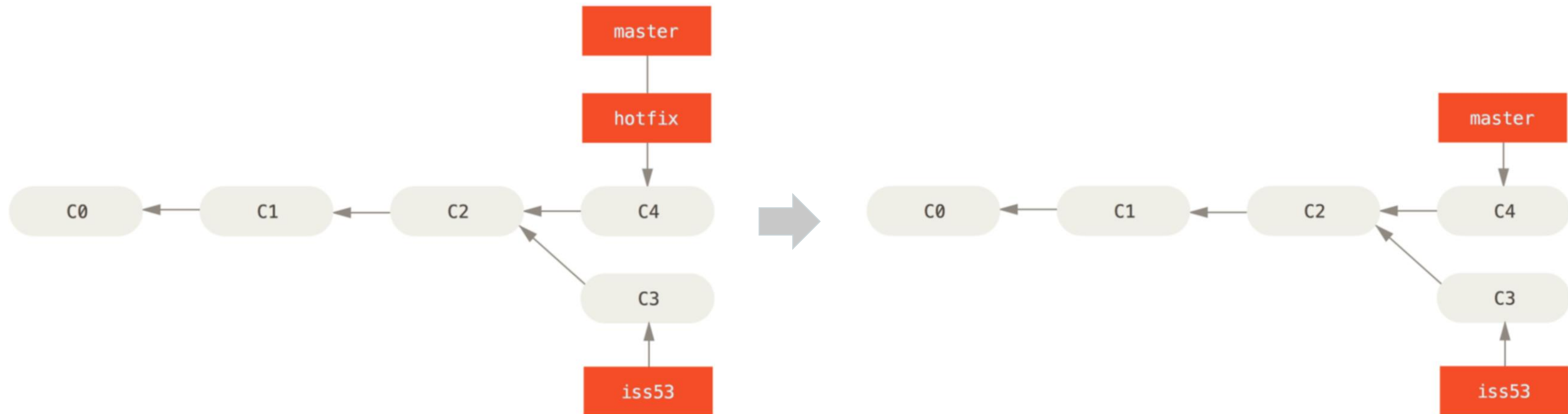
In the hotfix branch we run some tests and once we are get what we want, we can merge it back into the master branch.

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

git merging

Now we can delete the hotfix branch, because no longer needed.

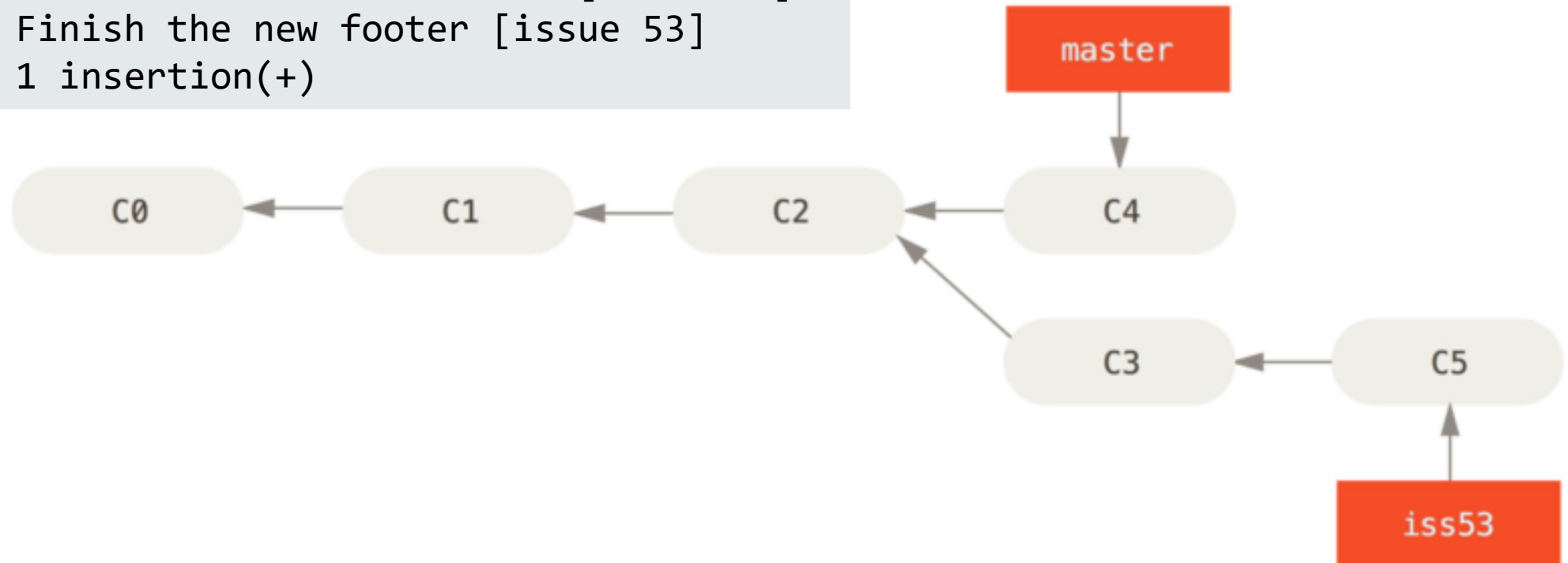
```
$ git branch -d hotfix  
Deleted branch hotfix (3a0874c).
```



git merging

Let's switch back to the work-in-progress branch on issue #53 and continue working on it.

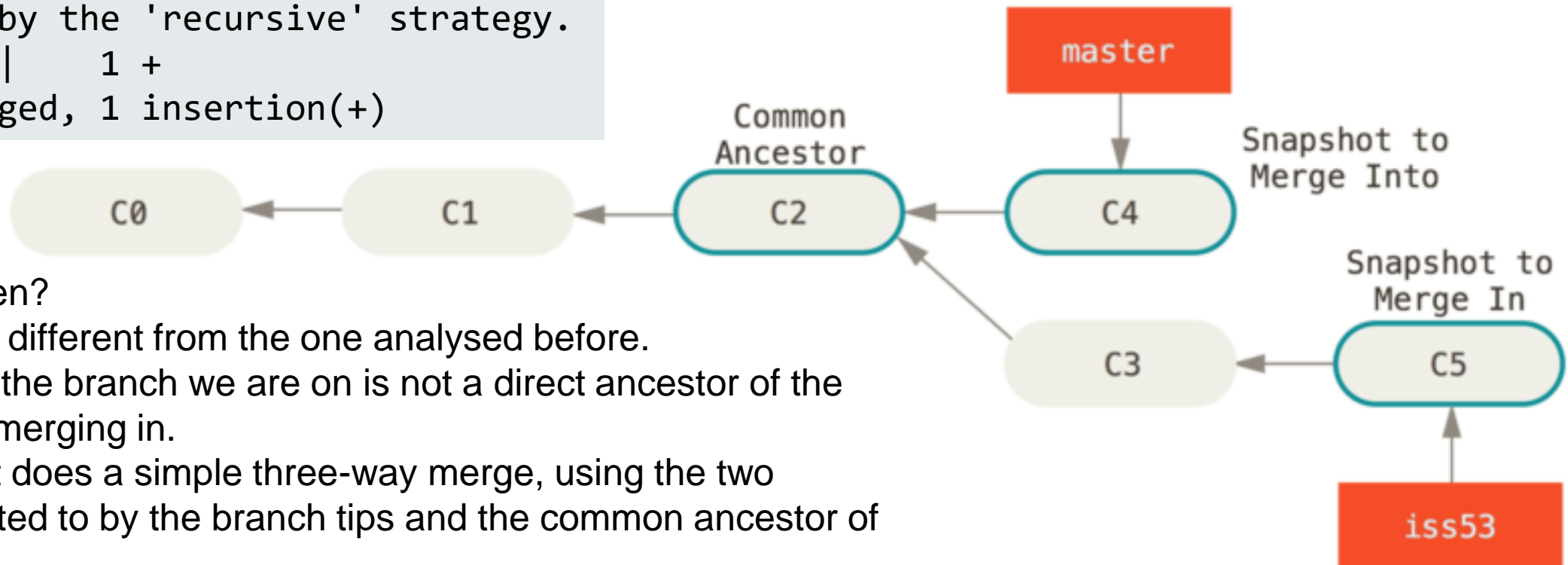
```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'Finish the new footer [issue 53]'
[iss53 ad82d7a] Finish the new footer [issue 53]
1 file changed, 1 insertion(+)
```



git merging

The work on issue #53 is complete and ready to be merged into the master branch.

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |      1 +
1 file changed, 1 insertion(+)
```



What will happen?

The scenario is different from the one analysed before.

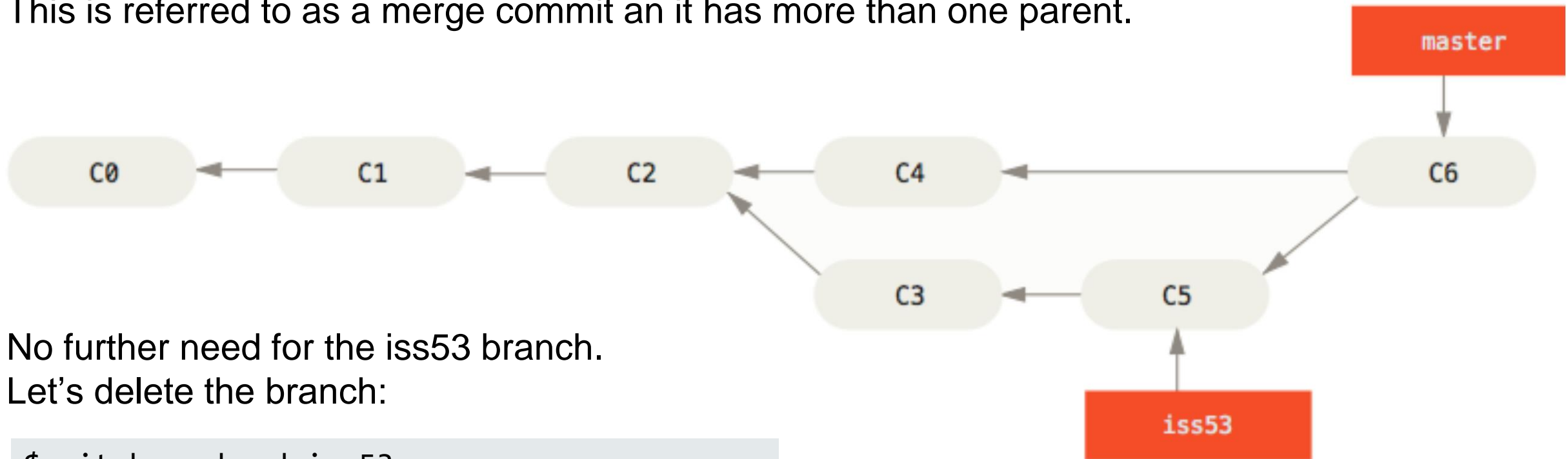
The commit on the branch we are on is not a direct ancestor of the branch we are merging in.

In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

git merging

Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it.

This is referred to as a merge commit as it has more than one parent.



No further need for the iss53 branch.
Let's delete the branch:

```
$ git branch -d iss53
```

git merging: basic conflicts

Occasionally, the merge process does not go smoothly.

If the same part of the same file is changed differently in the two branches we are merging together, Git will not be able to merge them cleanly.

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git does not automatically create a new merge commit, it pauses the process and let the user solve the conflicts, manually or using a tool, like mergetool (`$ git mergetool`).

To verify which files are unmerged → `git status`

Anything that has merge conflicts and has not been resolved is listed as unmerged.

Git adds standard conflict-resolution markers to the files that have conflicts to notify them to the user.

When all conflicts are over, then `git commit` and update the message.

Git on a server

In order to do any collaboration in Git, a remote Git repository (“Git server”) is needed.

A remote repository is generally a bare repository

a Git repository that has no working directory or simply a bare repository is the content of the project's .git directory and nothing else.

Git can use four major network protocols to transfer data:

- Local
 - **Secure Shell (SSH)**
 - Git
 - HTTP
- Obviously Git requires to be installed on the server.

Git on a server: repository

To create an empty repository

An empty bare repository on the (ssh) server can be initialized by issuing the following commands:

```
$ ssh user@git.example.com  
$ cd /opt/git $ mkdir my_project.git  
$ cd my_project.git  
$ git init --bare --shared
```

To initially set up any Git server

- Export an existing repository into a new bare repository

```
$ git clone --bare my_project my_project.git
```

- Put the bare repository on a server (scp → secure copy)

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

To let other SSH server users clone the repository

```
$ git clone user@git.example.com:/opt/git/my_project.git
```