Git Plumbing

- https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain
- User friendly commands are the **porcelain**
- Inner work commands are the **plumbing**

The Heart of Git

- Git is a **content addressable file-system**

A Typical Filesystem

```
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 3_Way_Merge
drwxr-xr-x  4 mattford  staff   128 Jun 21 13:57 Fast_Foward_Merge
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 Merge
drwxr-xr-x  4 mattford  staff   128 Jun 21 13:57 Rebase
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 Reset_Hard
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 Resetting_Hard
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 Resetting_Mixed
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 Simple_Conceptual_Model
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 The_Final_Picture
drwxr-xr-x  3 mattford  staff    96 Jun 21 13:57 Working_the_Diagram
-rw-r--r--  1 mattford  staff 11778 Jun 21 13:57 git-deep-dive.org
-rw-r--r--  1 mattford  staff  9842 Jun 21 14:41 git-plumbing.md
```

- Key Value pairs where the key (filename) looks up the value(content) of the file.

A Content Addressable File-system

```
"hello world"     -> hello world
"lots of data..." -> lots of data...
"ab345...def34"   -> ab345...def34
```

- We lookup the contents of a file using the files own contents!
- Why do we want to do this?

The Tao of Content Addressable File-systems and VCS's

- A Version Control System (VCS) wants to know if the file it's looking at is already stored in the VCS's persistence layer.
- What better way to do that than using it's own content as a key for the lookup? This is simply set membership.
- But what if the content (key) is very long? We still want efficiency and speed...

Digests to the Rescue!

- Digests are the outputs of class of functions know as hashes.
- A hash function takes arbitrary data of any size and returns data of a fixed size (the digest).
- It is a "function" in the mathematical sense of the word i.e, given the same data input you will always get the same digest.
- It may be the case that two different sets of data produce the same digest (but let's forget about this)

With Digests

Content Addressable file-system:

```
"Hello World!" -> "Hello World!"
```

With

```
hash("Hello World!") = "XJKT"
```

Becomes

```
"XJKT" -> "Hello World!"
```

Why are you so good Mr Digest?

- Hash functions allow any file content of any size to be represented by a digest.
- Assuming our VCS uses a content addressable file-system as its back-end, then every time the VCS wants to know if a file changes, it calculates the digest and looks it up in the backend.
    - if the file exists then it's unchanged
    - if the file doesn't exist then it has changed
- This much quicker than using diff'ing algorithms to work out if there's been a change or not. These calculations can be **deferred** until when the user needs them.

So many good Digest things

- What other advantage is there of using content as the lookup key?
    - renaming?
    - deduplication?
- Git uses, by default, SHA-1 has the hashing function
- Our first plumbing command!

```
echo 'Hello World!' | git hash-object --stdin
```

- Without options `hash-object` just returns the digest

Writey, write, write

- `hash-object` has more options. We can write content to backend/db/persistence-layer/content-addressable-file-system by passing `-w`.

```
echo 'Hello World!' | git hash-object -w --stdin
```

- Where does this now live?

```
ls -l .git/objects/98/0a0d5f19a64b4b30a87d4206aade58726b60e3
```

- All content for git lives in objects in the `.git/objects/` directory.
- The naming scheme is simple:
  - first 2 digits of the hash form a directory
  - remaining 38 digits are the name of the object(file)

Ready, read, read

- We can fetch the data out the db with

```
git cat-file -p 980a0d5f19a64b4b30a87d4206aade58726b60e3
```

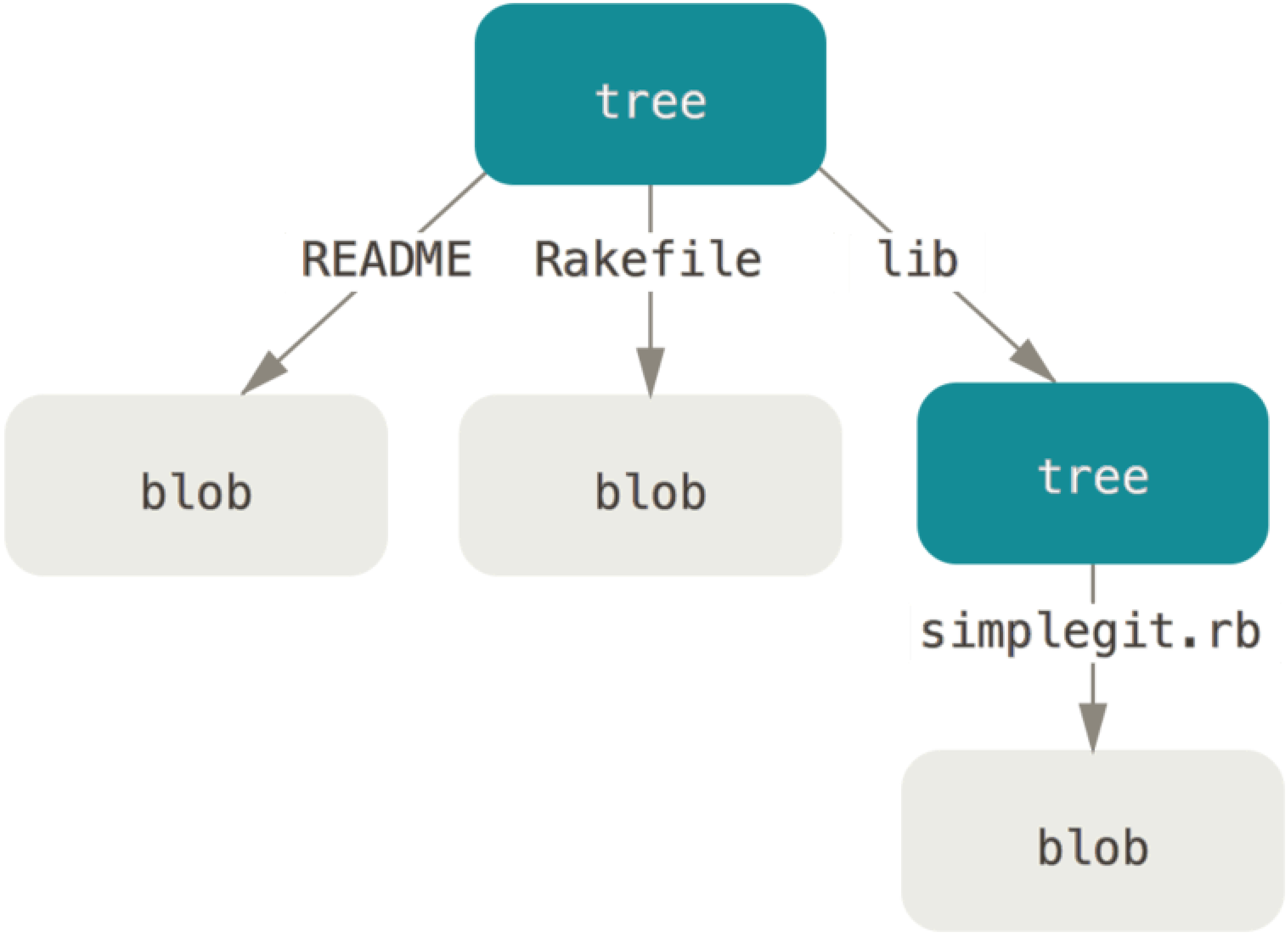
- And view the data with

```
ls -l .git/objects/* | head -n 15q
```

Seeing the wood for the trees

- What's missing now we can read and write all our content?
- How do we store file names and file-system hierarchy? Meta-data in general?
- Tree objects do this!

Simple Conceptual Model



A Real Tree

```
git cat-file -p main^{tree} | head -n 10
```

- Notice the columns
  `permissions : type : sha-1 : file-name`

Growing (making) a Tree (object)

- The "index" or "staging area" is a essentially the content of a tree object that's being built up whilst we work.
- With this in mind We now have enough machinery to begin to talk about what happens when we stage (git add) files:
    - write the content object to the back-end
    - add to the index (the tree object to be) the data about the content

Plumbing git add

```
echo "JUXT forever" | git hash-object --stdin -w

git update-index --add --cacheinfo 100644 \
  0a67bfca9b837c46c80e9631d7407e496878173b juxt.txt

$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579

$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 0a67bfca9b837c46c80e9631d7407e496878173b  juxt.txt
```

An Object Still Missing

- We've added our content to the persistence layer.
- We've added to the index the reference to the content and given it a filename, and permissions (it's metadata).
- (we optionally loop here)
- We've written the index as a tree object to the persistence layer.
- Potentially we repeat the whole process.

What do we have?

- There are now lots of trees...

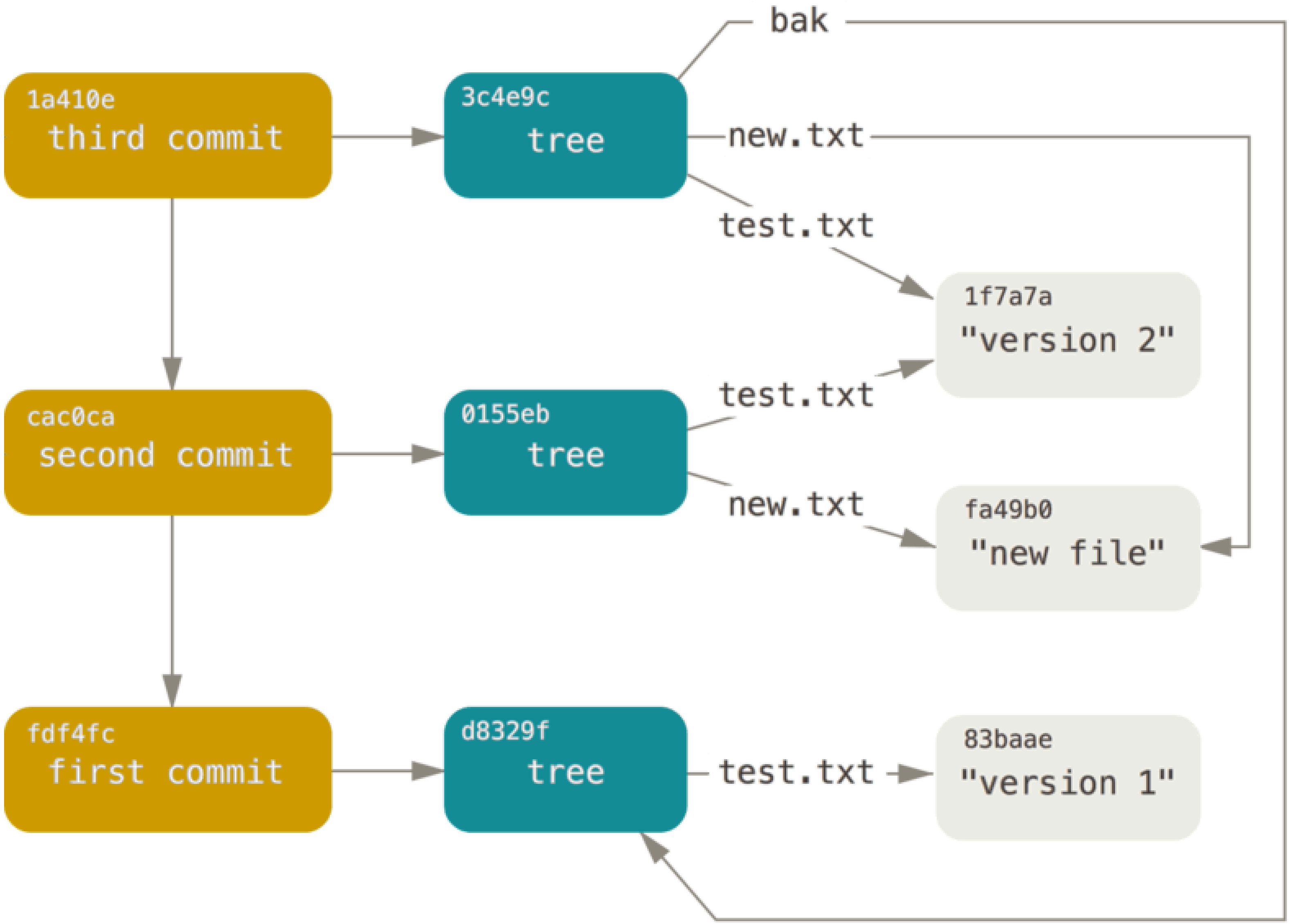
What are we missing?

- There's now lots of unordered, unrelated trees.

Commit's to the rescue

- If we had an ordering of the trees we've been creating in our update loop - what would we have?
- A history of state! Snapshots of the working directory over the changes.

Working the Diagram



What else can a Commit do for us?

- Considering a VCS what other features are required?
    - Who?
    - When?
    - Why?
    - Basically work related Metadata
- A commit object does all this! The parent reference provides the ordering.

What does a Commit look like?

```
$ git log

$ git cat-file -p 7c721
tree 236a0d5ad63e7b6883d40e843b30ebbc374d6acf
author Matt Ford <matt@dancingfrog.co.uk> 1718981594 +0100
committer Matt Ford <matt@dancingfrog.co.uk> 1718981594 +0100

First pass
```

Is there anything another layer of indirection can't solve?

- Commit objects are great'n'all but they are not very friendly.
- We want a humane way of talking about commits

```
find .git/refs
```

- Compare with

```
git branch
```

But what do these refs contain?

```
cat .git/refs/heads/master
```

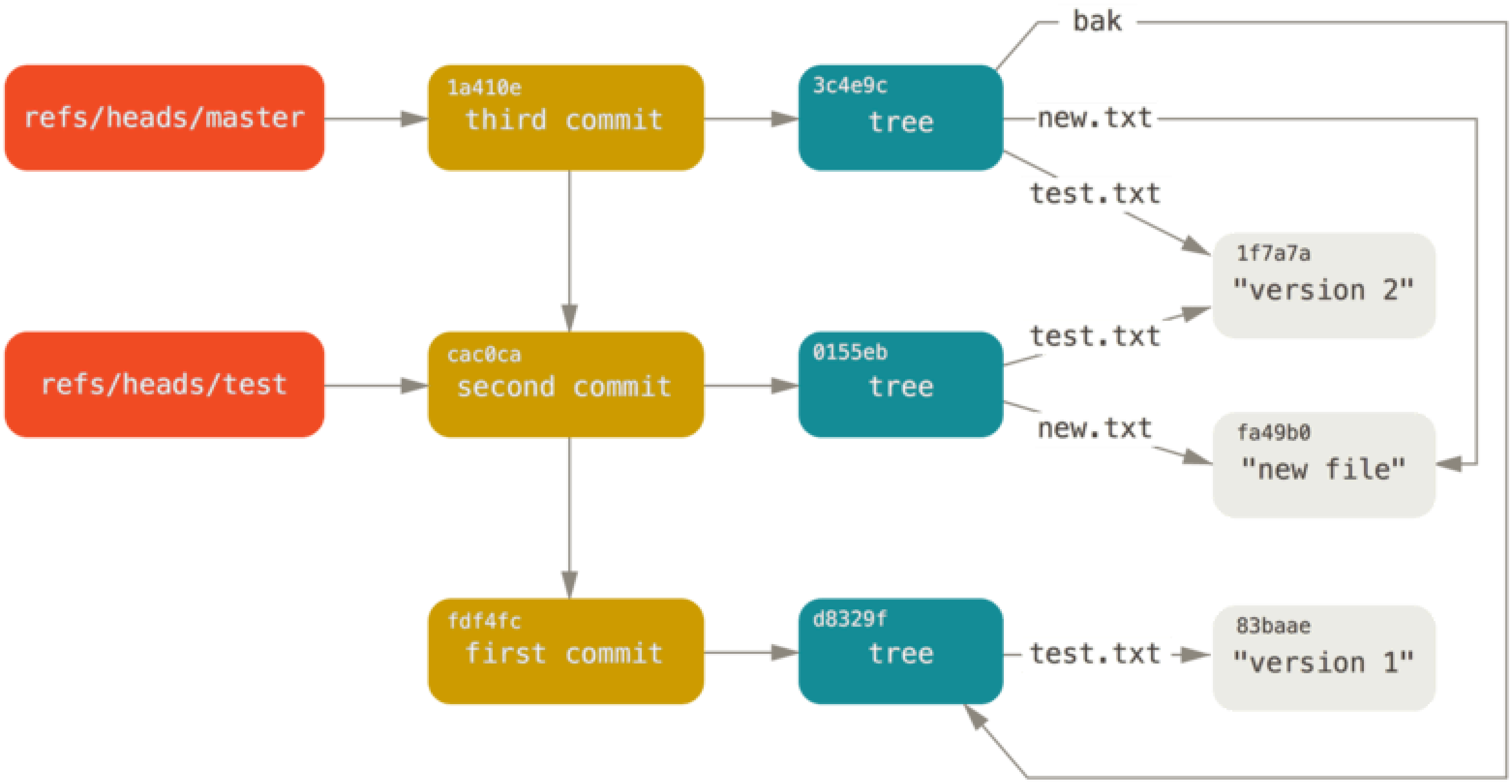
- So a ref in the `heads` folder is named for the local branch it represents.
- It's contents are the digest of the Commit object it points to.

You can unofficially create a ref with something like

```
echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

where the digest points to a commit object.

The Final Picture



Refs and branches

- The official way to create a ref is

```
git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

- What happens when we create a branch?

```
git branch <branch>
```

- Git takes the digest of the last commit object and writes it to a ref file named as `<branch>`.
- BUT HOW DO WE KNOW THE LAST COMMIT OBJECT?

Yet another redirection!!

```
cat .git/HEAD
```

- it's a symbolic reference to another reference!
- to be detached: occasionally HEAD will not point to another reference but instead will contain a Commit Object digest (or perhaps a tag)
- it typically means you are some point in a branches history and not at HEAD.

Tag this, tag that.

- Tags provide a human friendly name to a point in time on a branch.
- There are two types of tags.
- Tag Objects or an annotated tag, like a commit object in function. Commit Objects point to a tree whereas a Tag object points to a commit. A reference is created that points to the Tag object.
- Tag reference or lightweight tag: no tag object is created but a tag reference is created that points to the commit.

Summarize this!

- central concept: content-addressable-file-system
- content objects: addressable by digest
- tree objects: grouping content objects
- an index structure (staging area) containing references to tree and hash objects
- commit objects: provide an ordering by parent relationship, point to tree objects, have metadata
- refs: files, friendly named, that contain commit objects digests
- HEAD a reference to a ref, keeps track of the current commit
- tags: pointers to specific commits.

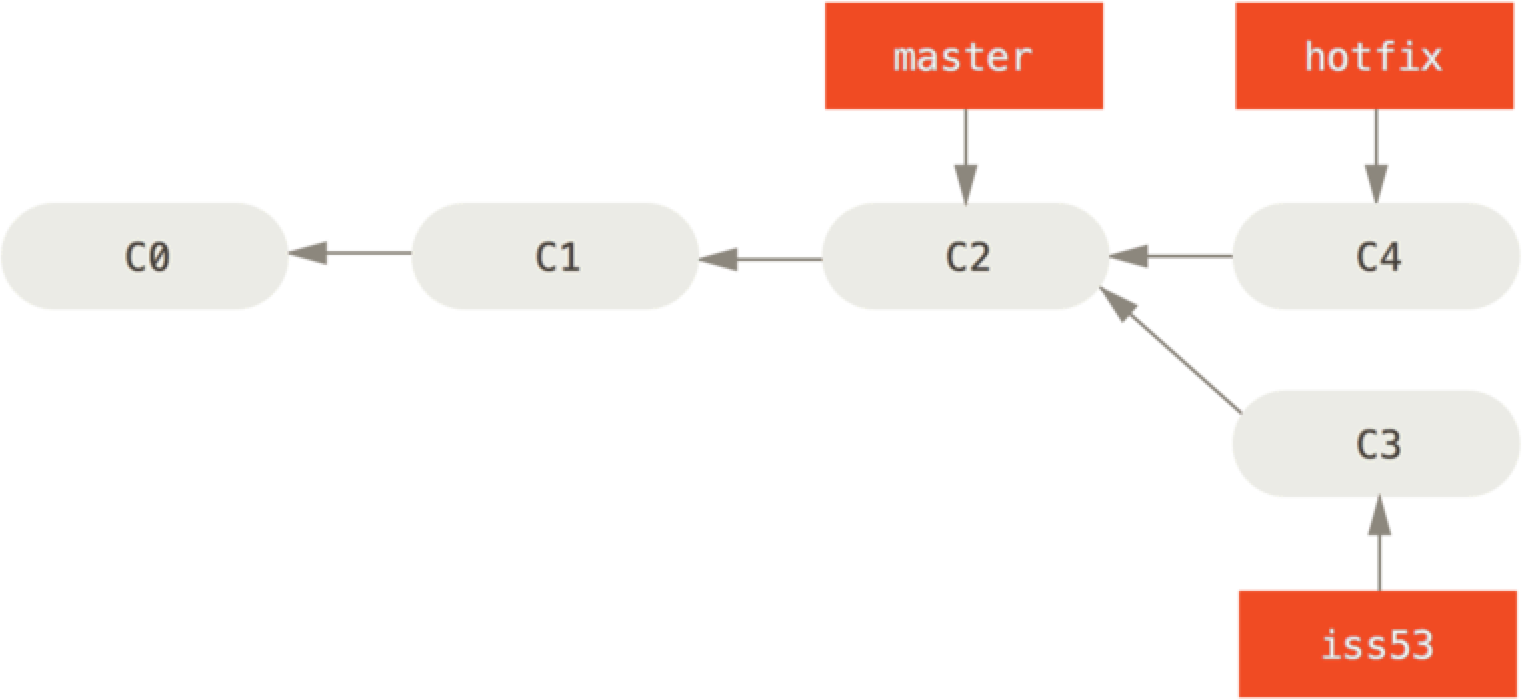
Together time

- How does our model work with common operations?
- We've covered:
    - staging
    - commit
    - branching
    - tags

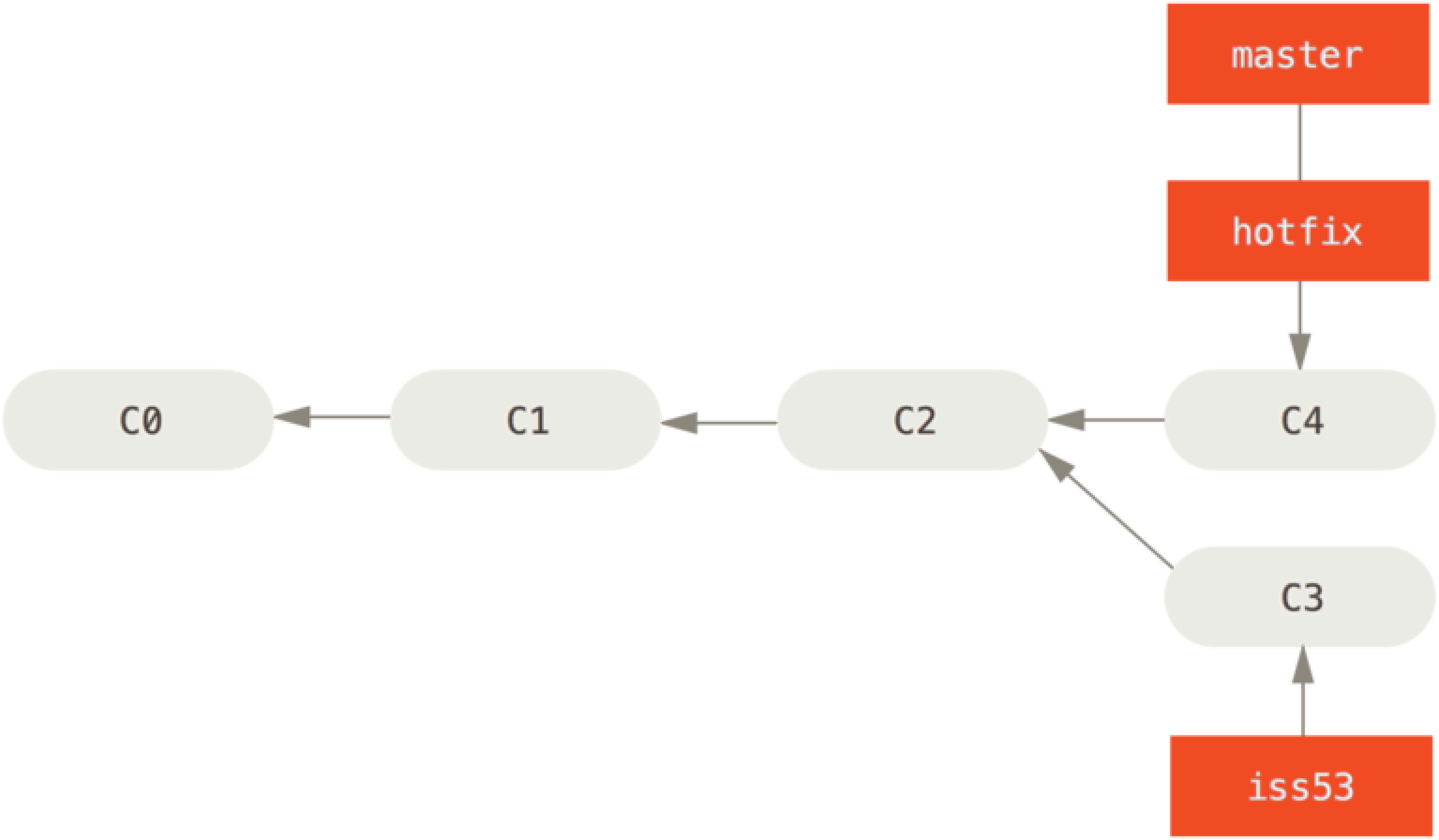
Diffing Commits

- What's the approach?
- What we can we reason out?
- Analyse the tree structure

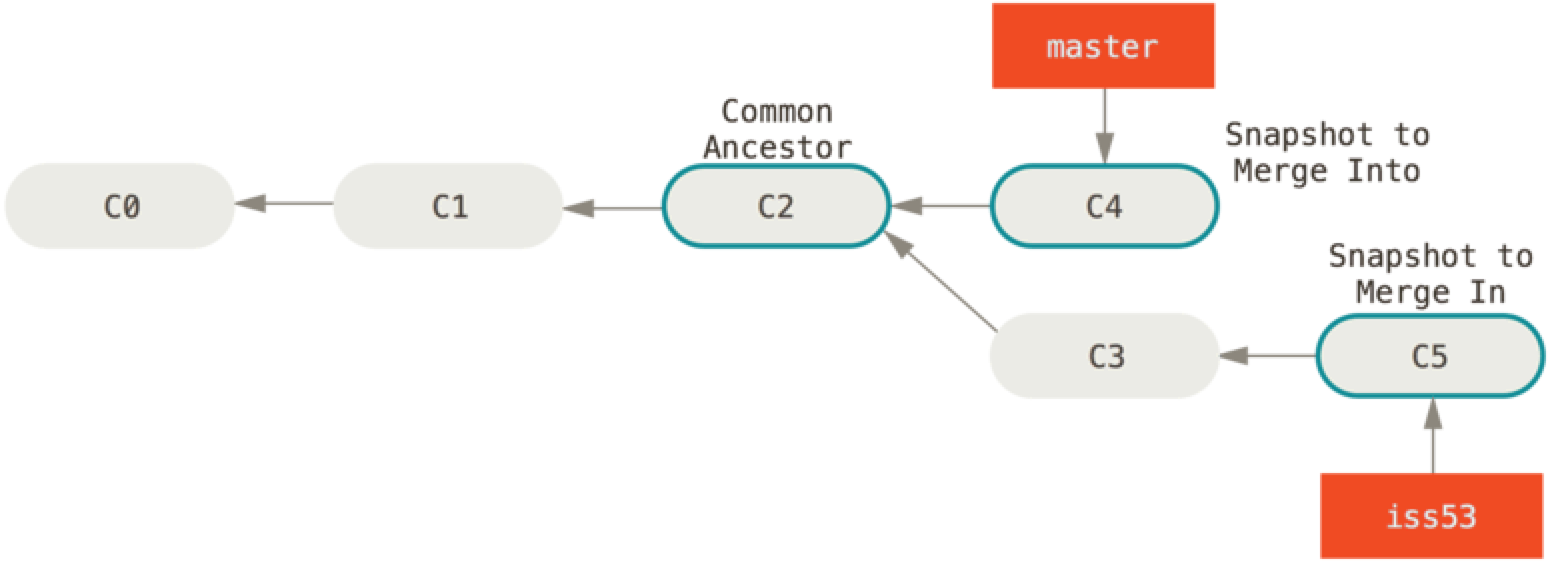
Fast Forward Merge (1)



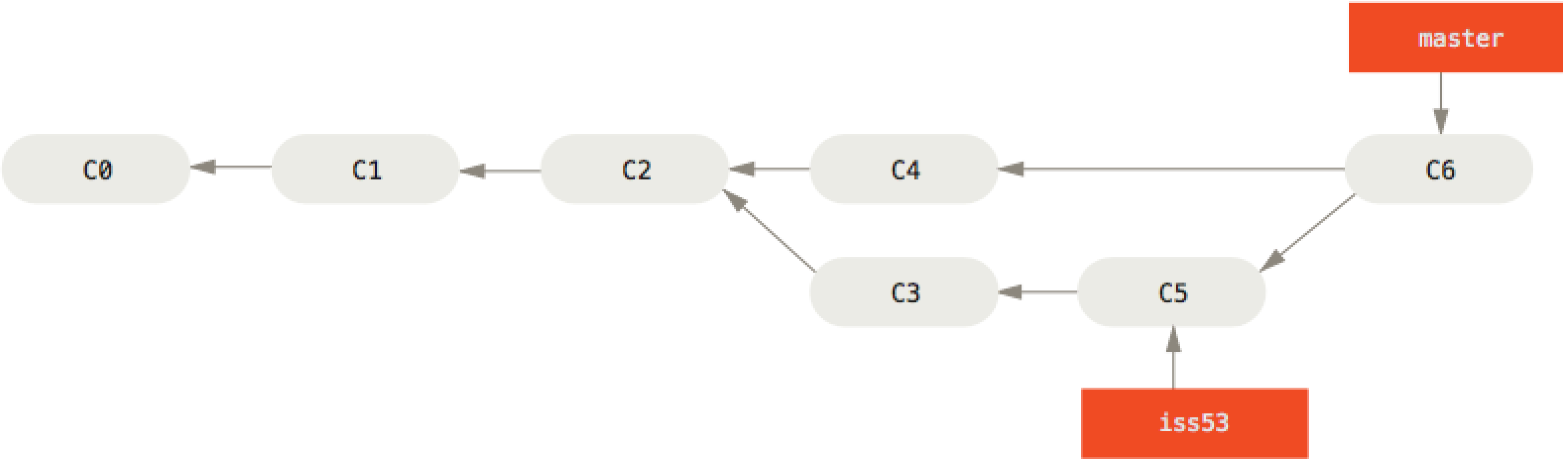
Fast Foward Merge (2)



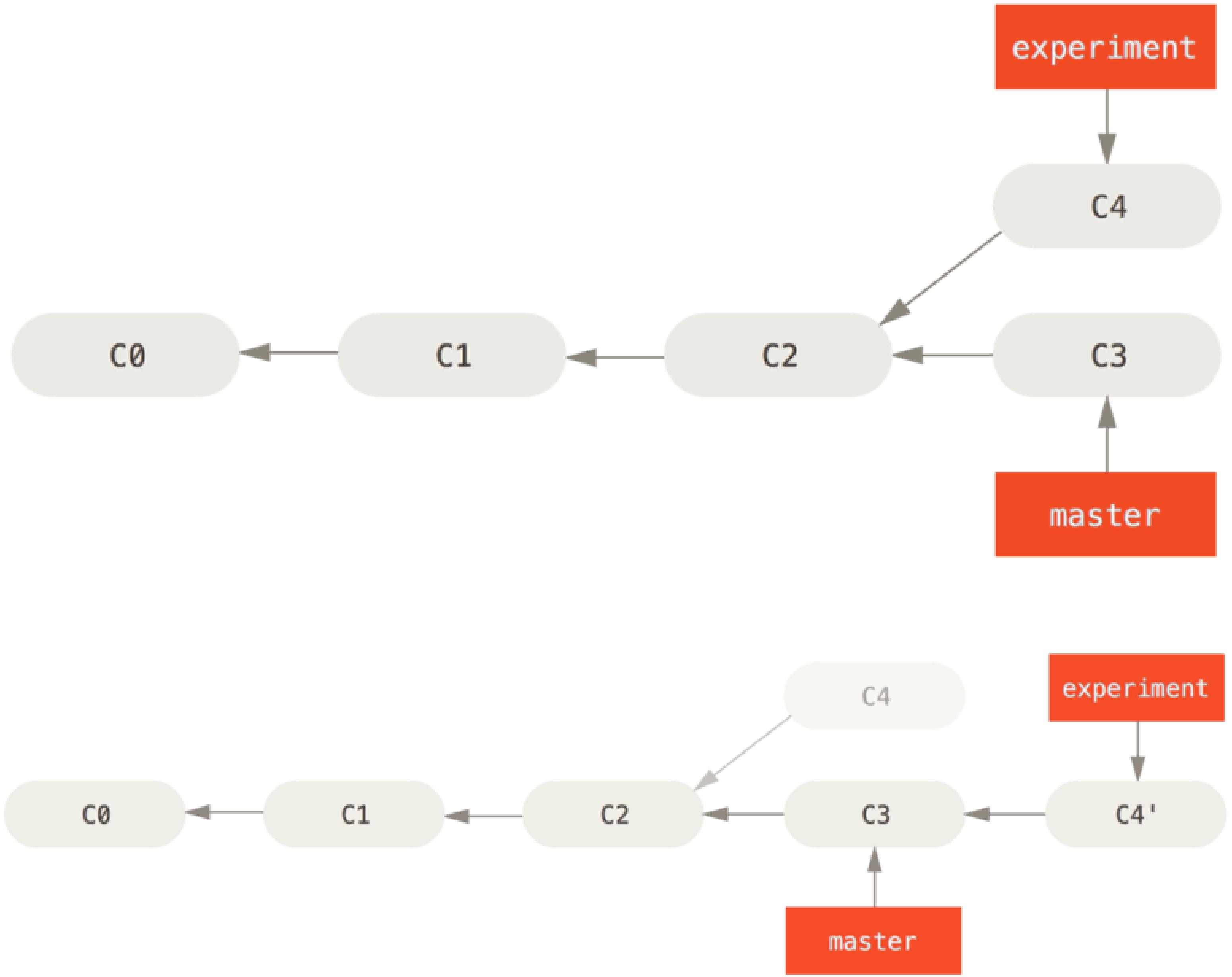
3 Way Merge (1)


master
Common
Ancestor
Snapshot to
Merge Into
C0
C1
C2
C4
C3
C5
Snapshot to
Merge In
iss53

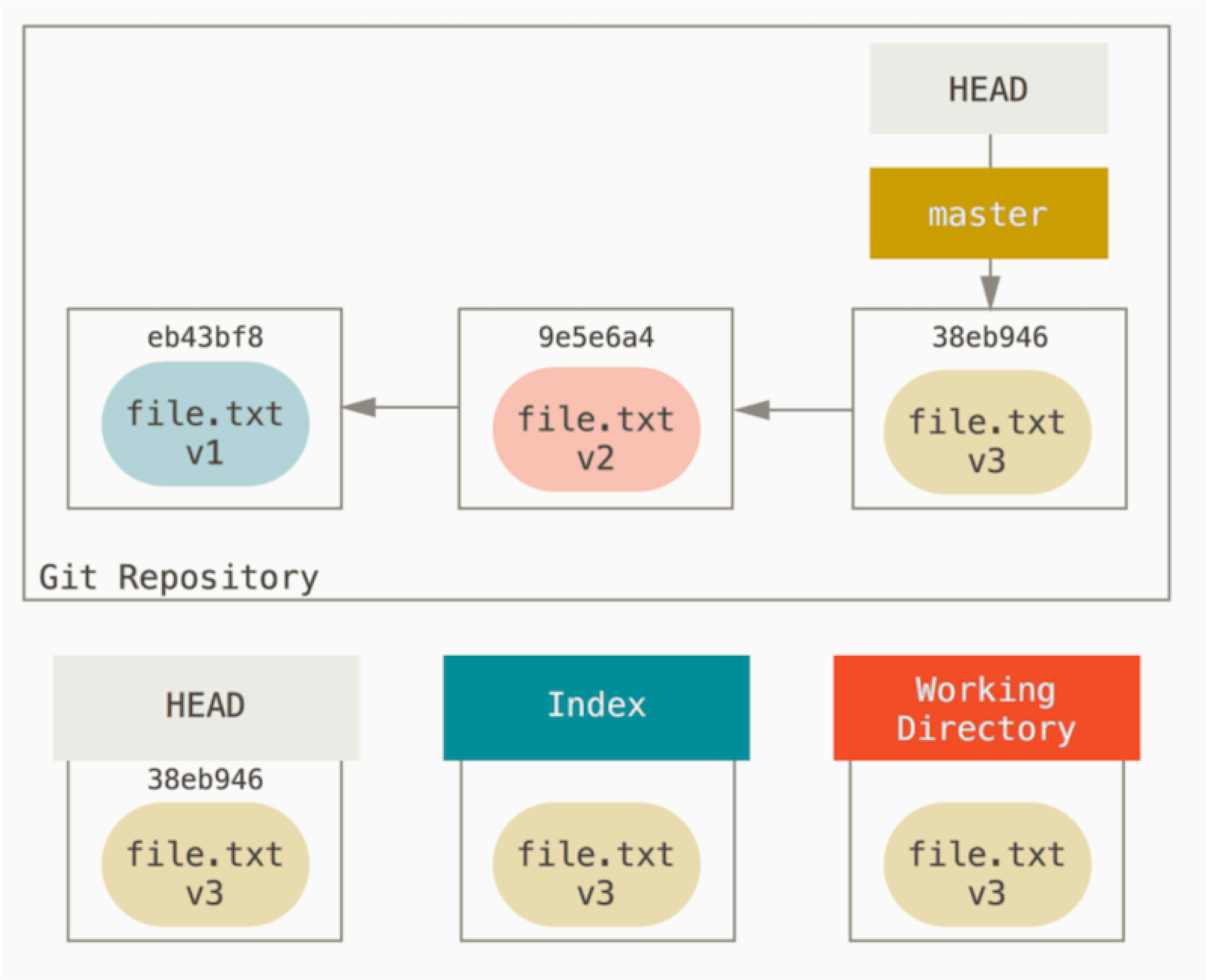
3 Way Merge (2)

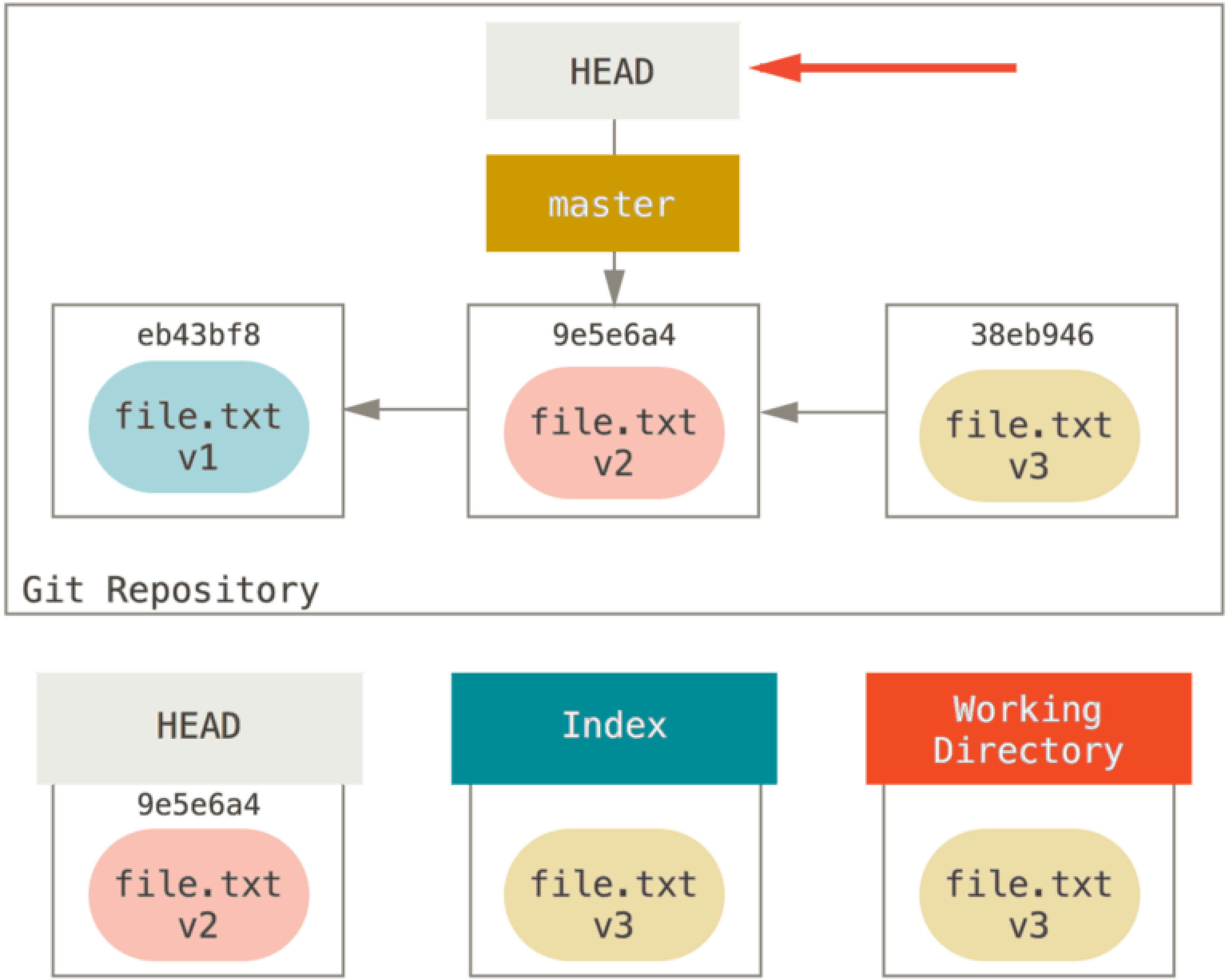


Rebase


experiment
C4
C0
C1
C2
C3
master
C4
experiment
C0
C1
C2
C3
C4'
master

Resetting


HEAD
master
eb43bf8
file.txt
v1
9e5e6a4
file.txt
v2
38eb946
file.txt
v3
Git Repository
HEAD
38eb946
file.txt
v3
Index
file.txt
v3
Working
Directory
file.txt
v3



HEAD

master

eb43bf8

file.txt
v1

9e5e6a4

file.txt
v2

38eb946

file.txt
v3

Git Repository

HEAD

9e5e6a4

file.txt
v2

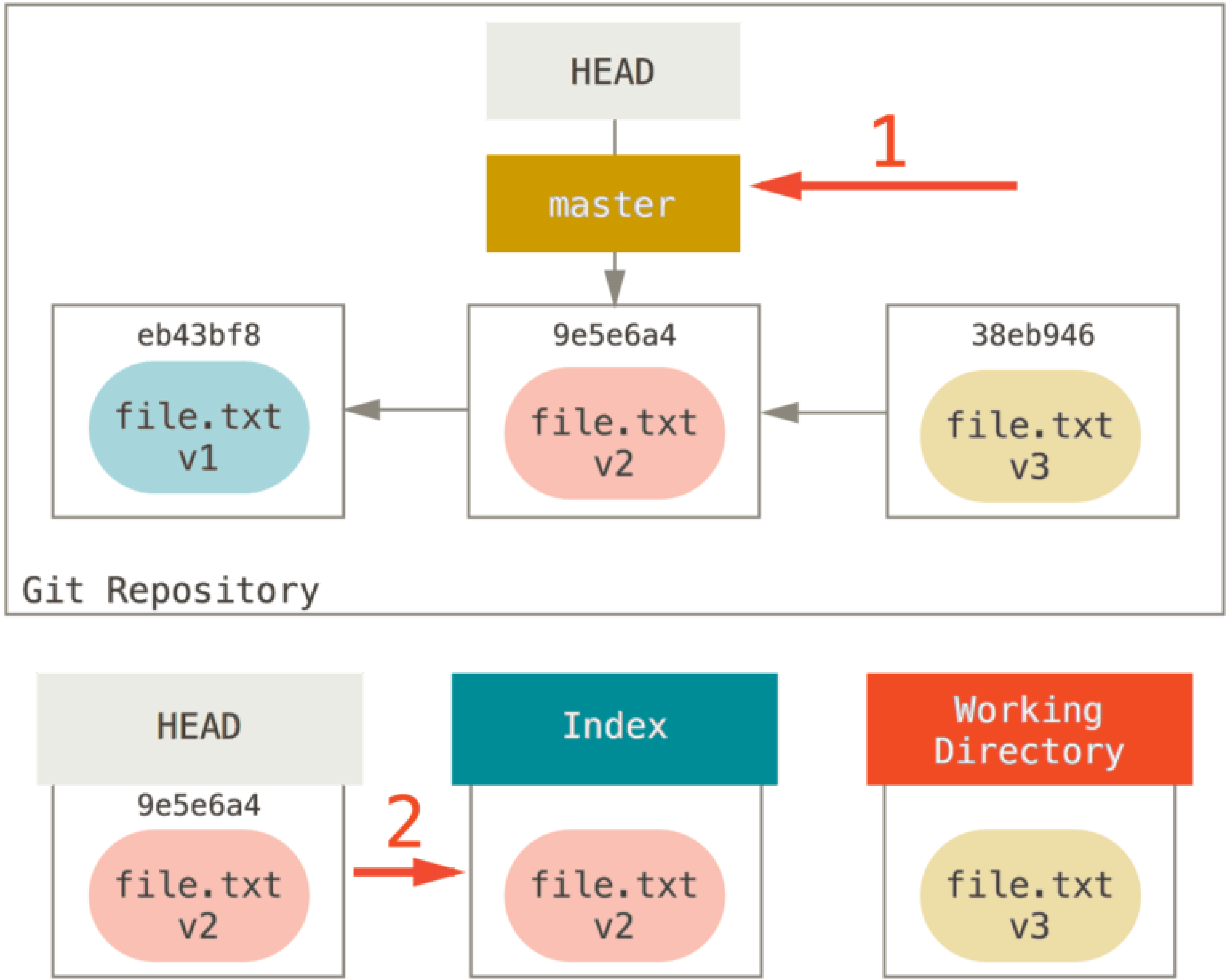
Index

file.txt
v3

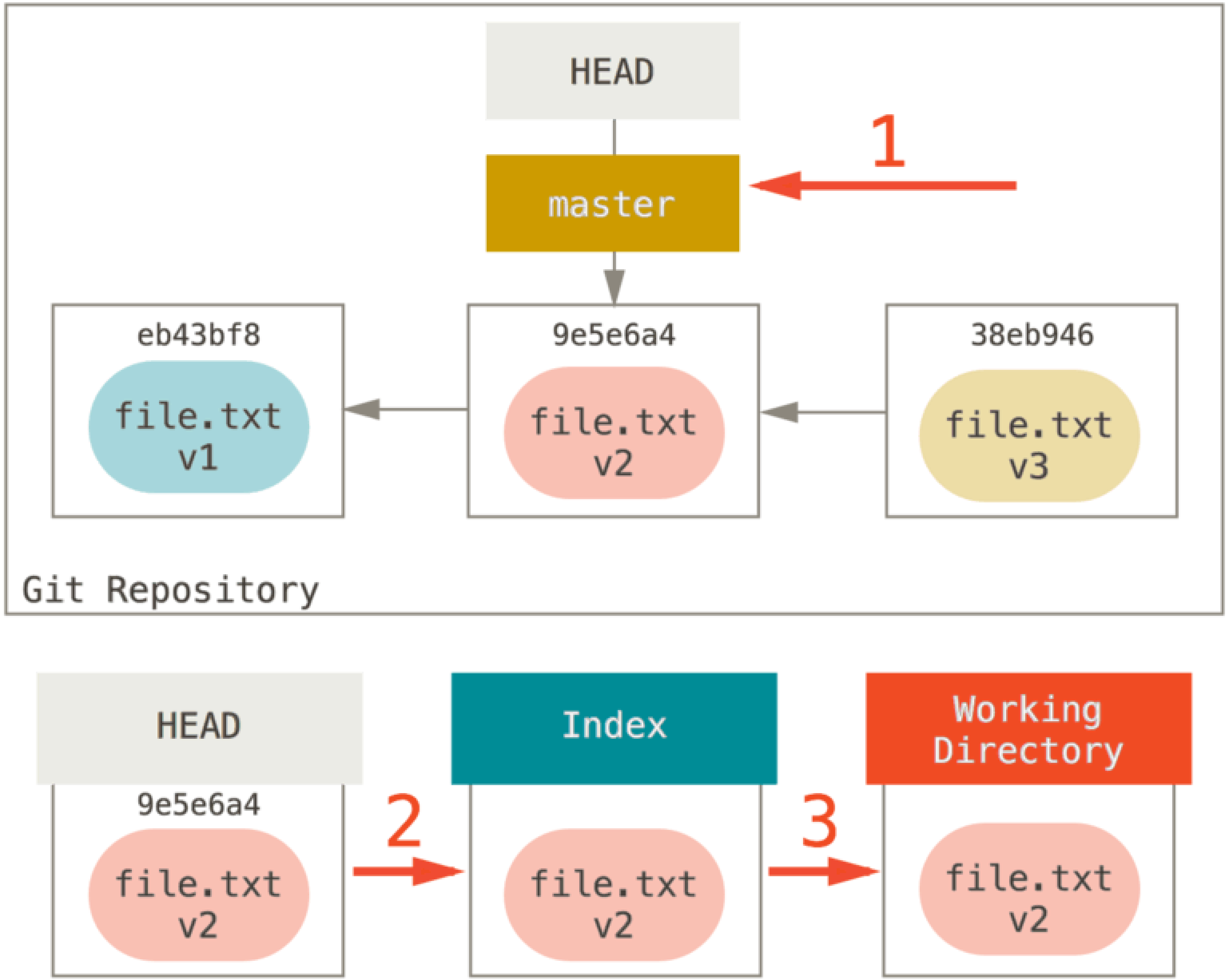
Working
Directory

file.txt
v3

git reset --soft HEAD~



```
git reset [--mixed] HEAD~
```



git reset --hard HEAD~