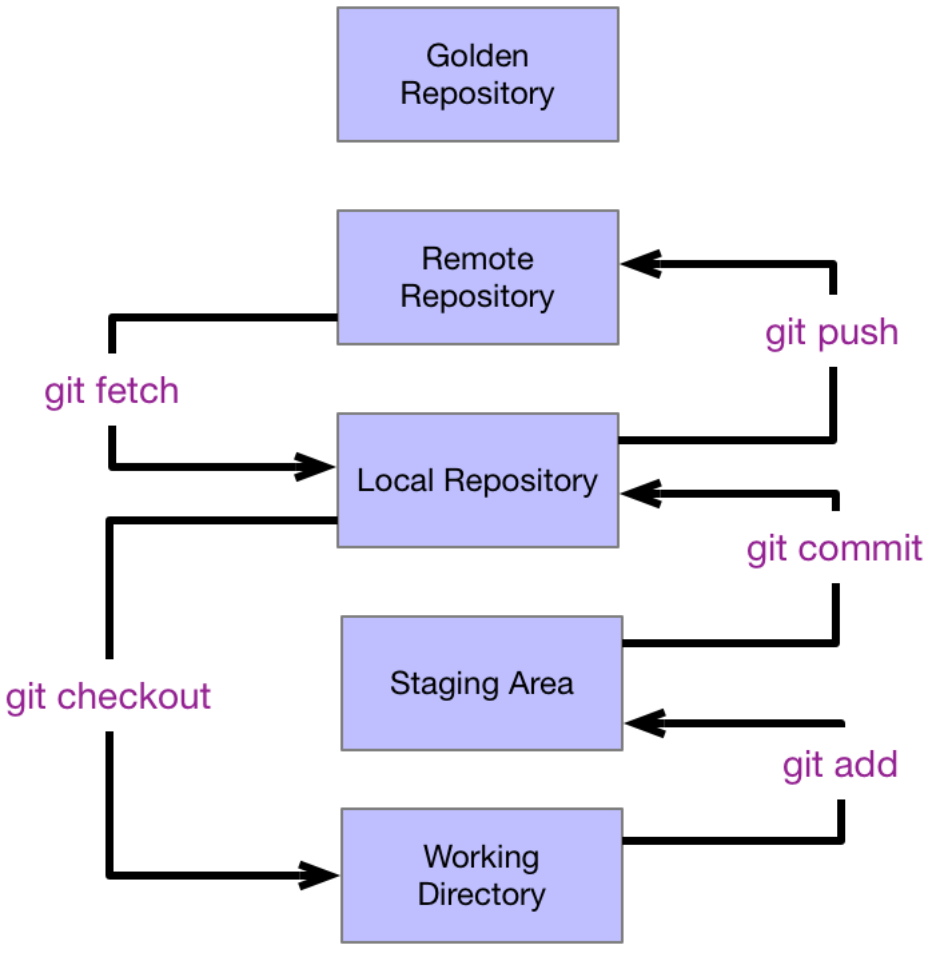
Kenny R N Wilson

Git

THIS DOCUMENT COVERS

A git projects exists in many layers. At the enterprise level we typically have five layers. The following diagram shows the layers and the commands used to move files between layers

Figure 1 Layers in Git



Repository Structure

CREATING A REPOSITORY

The directory into which we create a git repository is known as the **working directory**. To first setup a local repository we move to the directory which we want to be the working directory and issue the command

git init

Issuing this command causes git to create a repository in a subdirectory of the working directory called .git Let us look at the contents of the .git repository folder to see what it contains.

Figure 2 Empty Working Directory

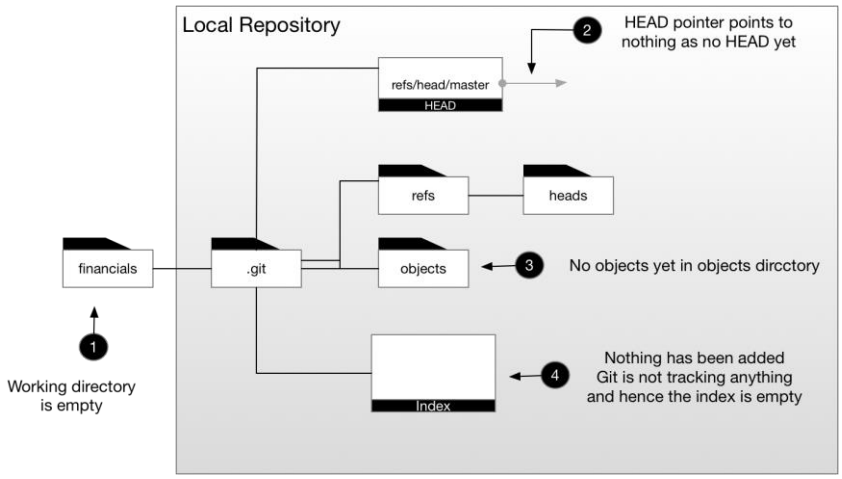


TABLE 1 REPOSITORY STRUCTURE

File/Directory	Description
HEAD	Points to (usually) the current commit on the current branch
objects	Contains the blob objects that make up staged files and commits (file, directory, commit)
refs	Anything that points to a blob via its SHA1 hash value. Includes branches and tags
index	Contains references to files in the staging area

ADDING CONTENT TO THE WORKING DIRECTORY

Starting from an empty working directory we create a single file

echo “Hello World” > hello.txt

Git does not know anything about this file. We say the file is **untracked**. Git will not start tracking a file until we tell it to do so. If we ask git to tell us the status via the command

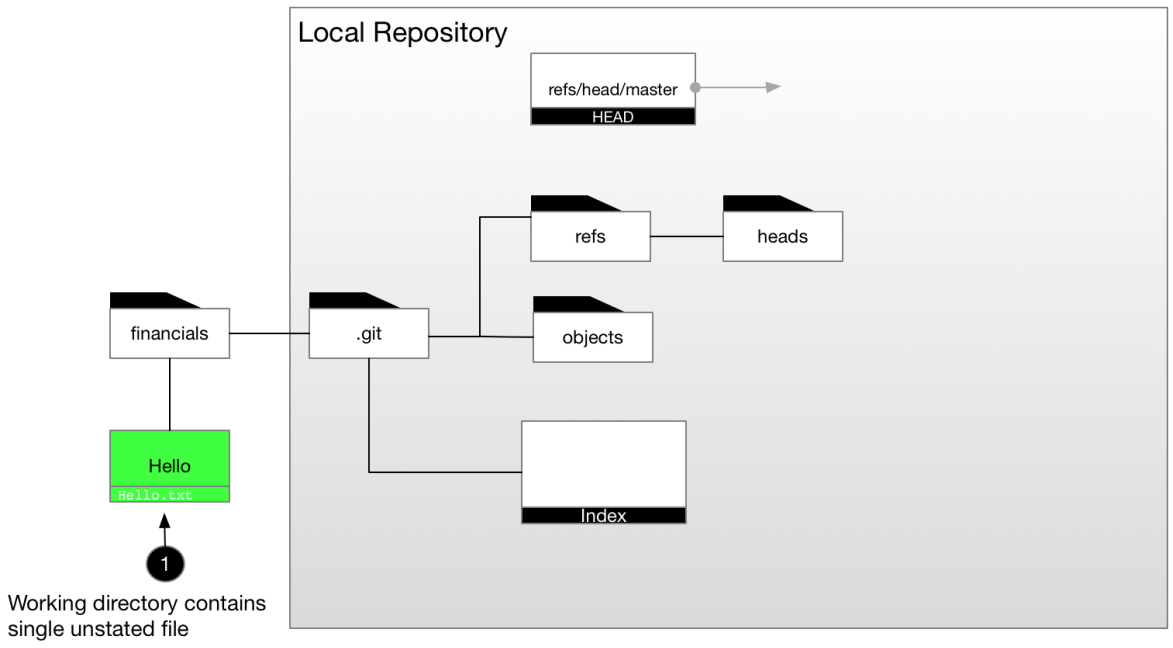
git status

which outputs the following

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        hello.txt

nothing added to commit but untracked files present (use "git add" to
track)
```

Our working directory and repository now looks as follows

Figure 3 Untracked File



STAGING A FILE

We now have a file in the working directory which is not being tracked by git. If we want git to track it then we need to add it to the staging area.

git add hello.txt

```
$ git status
On branch master

No commits yet

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

        new file:   hello.txt
```

Now our git repository knows about our file.

Figure 4 Staged File

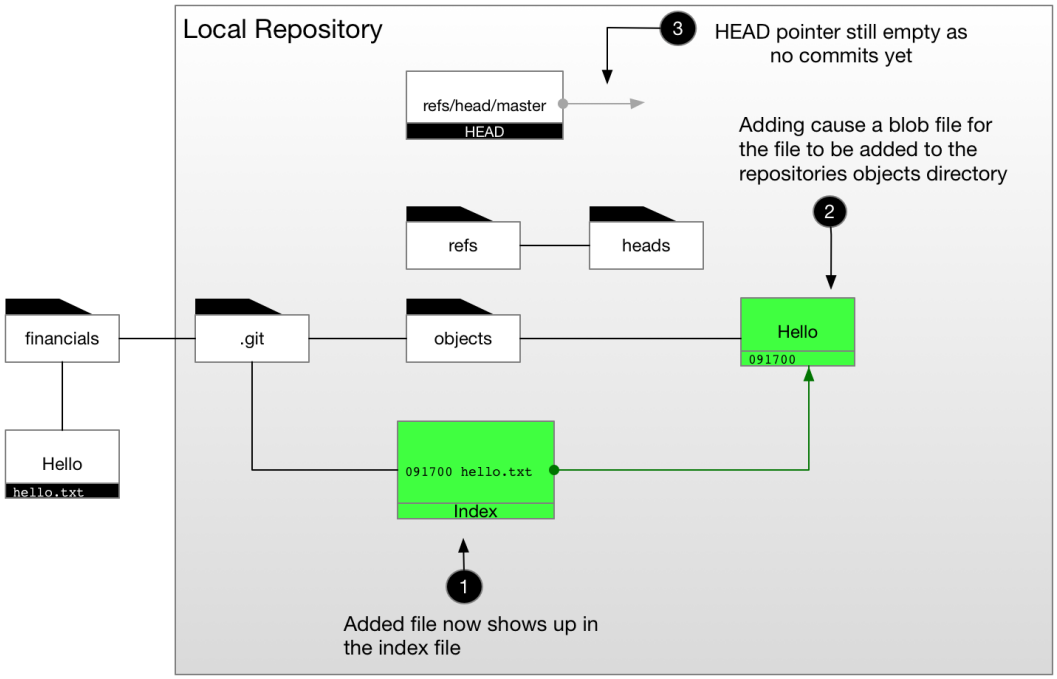


TABLE 2 COMMAND FOR EXAMINING REPOSITORY BLOBS

Command	Example
List Index File	git ls-files –stage
List Object Type	git cat-file -t 557db03
List File Objects Contents	git cat-file -p 557db03
Remove a file from index (staging)	git reset hello.txt

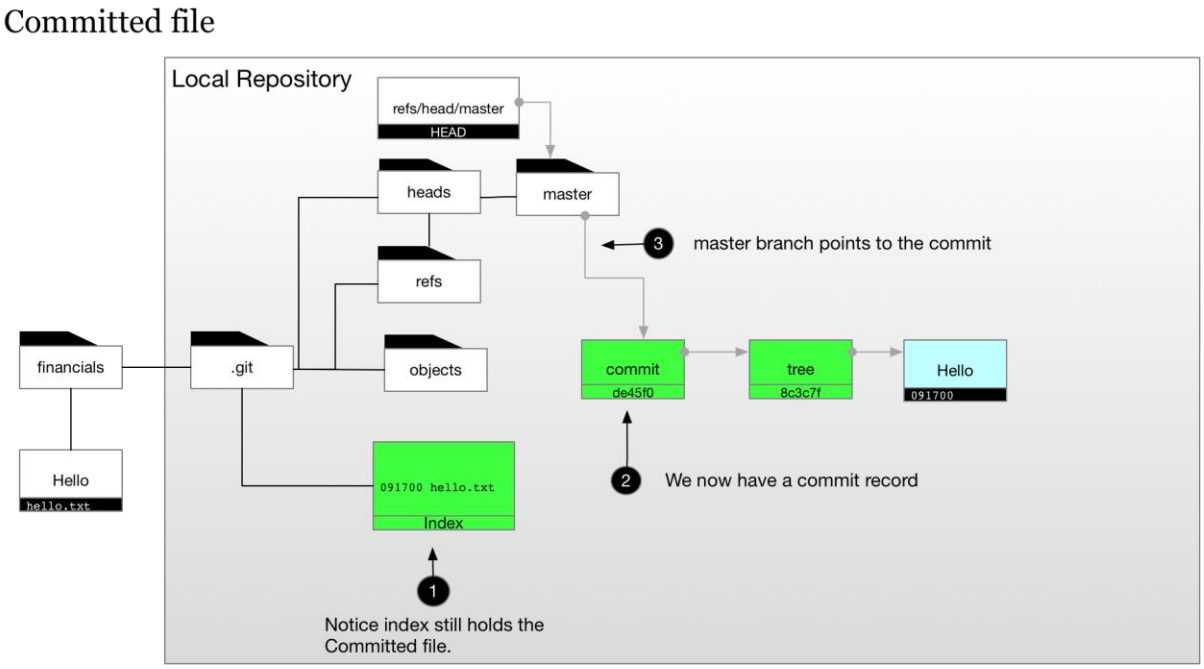
COMMITTING A FILE

From the staging area the next step is to commit our file to Git. We use the commit command.

git commit -m “First cut of hello” hello.txt

Now our .git directory structure is starting to take shape

Figure 5 Committed File



A SECOND COMMIT – COMMIT CHAINING

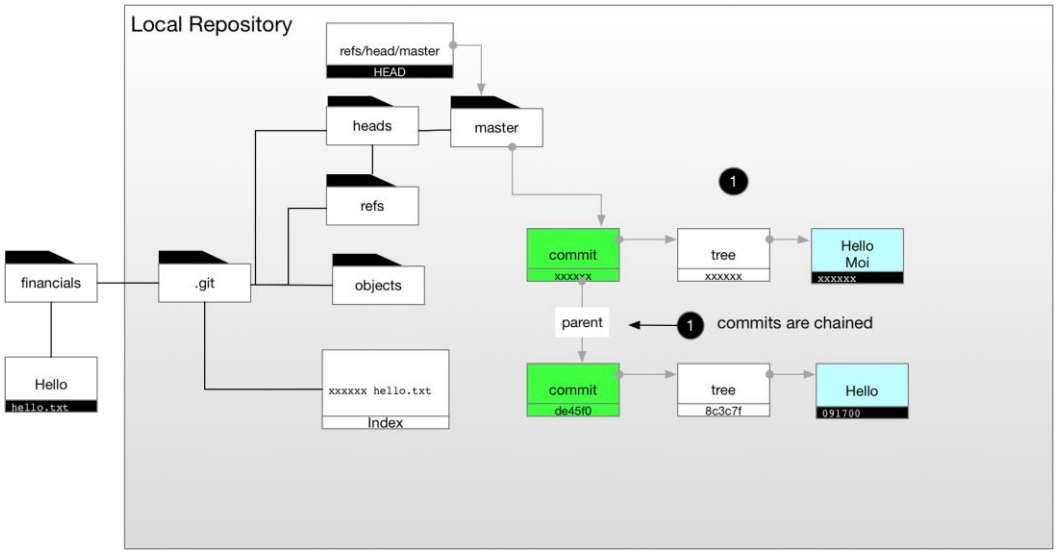
The key thing to note when adding a second commit is that it has a pointer back to the original commit. In this way commits are chained together

echo “Moi” >> hello.txt

git add hello.txt

git commit -m “second line in file” hello.txt

Figure 6 A Second Commit



Merging

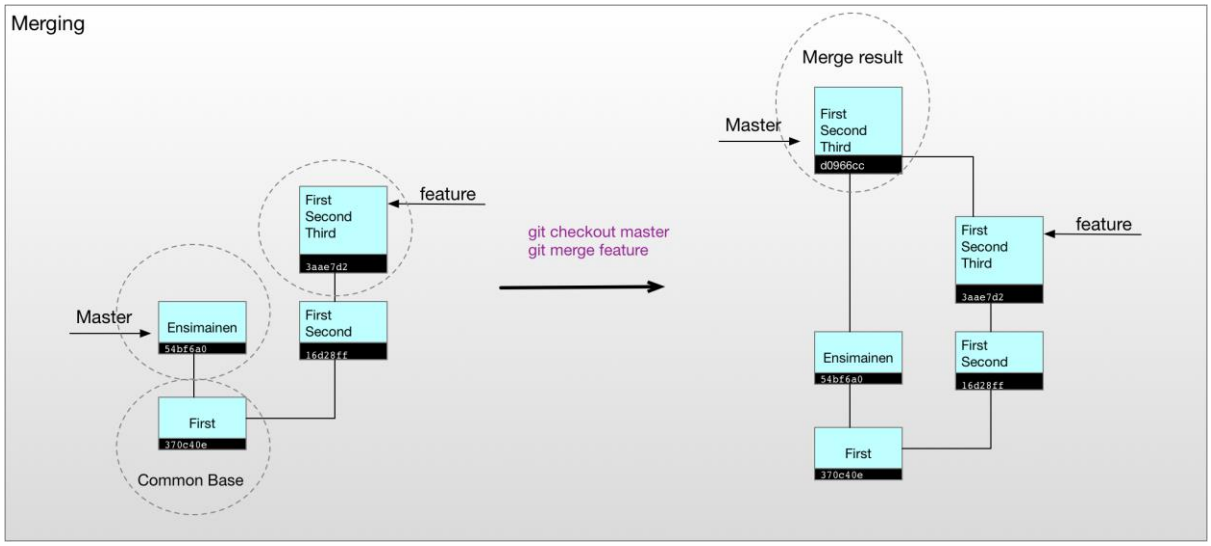
Merge

git merge merges a named branch into the current branch. The current branch is the target and the named branch is the source. If the destination branch has not had any changes since the source branch was created, we can do a fast forward merge. The following merges the source branch called feature into the destination branch feature.

git checkout master

git merge feature

Figure 7 Merging



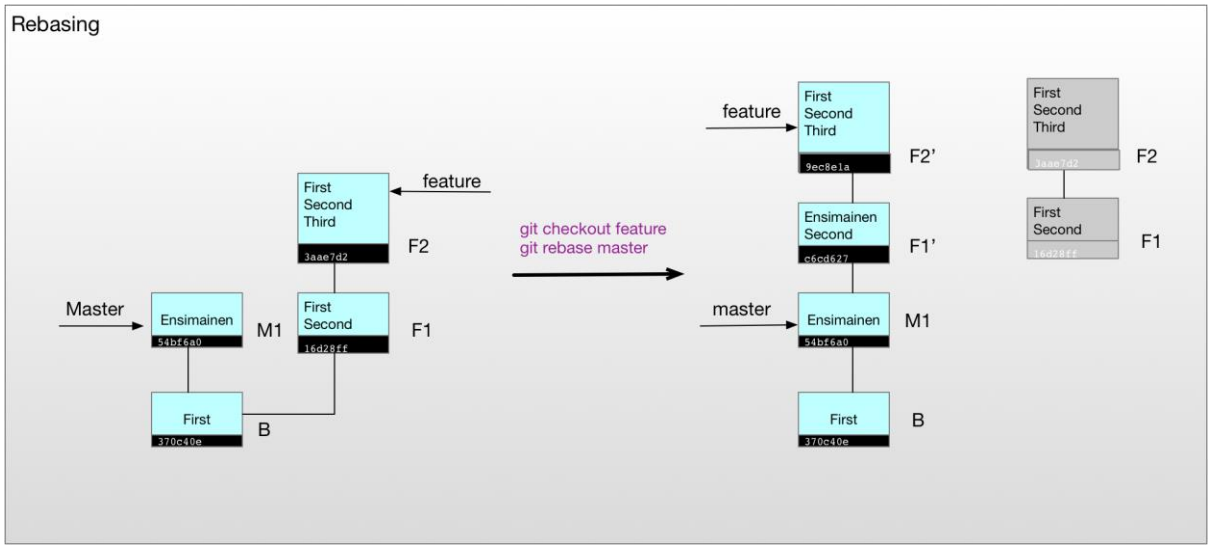
Rebase

Rebase takes the changes in *feature* and applies them on top of the head of *master* and then updates *feature* to point to resulting latest delta commit. Master is **unchanged** but now it can be easily fast forwarded to the tip of feature.

git checkout feature

git rebase master

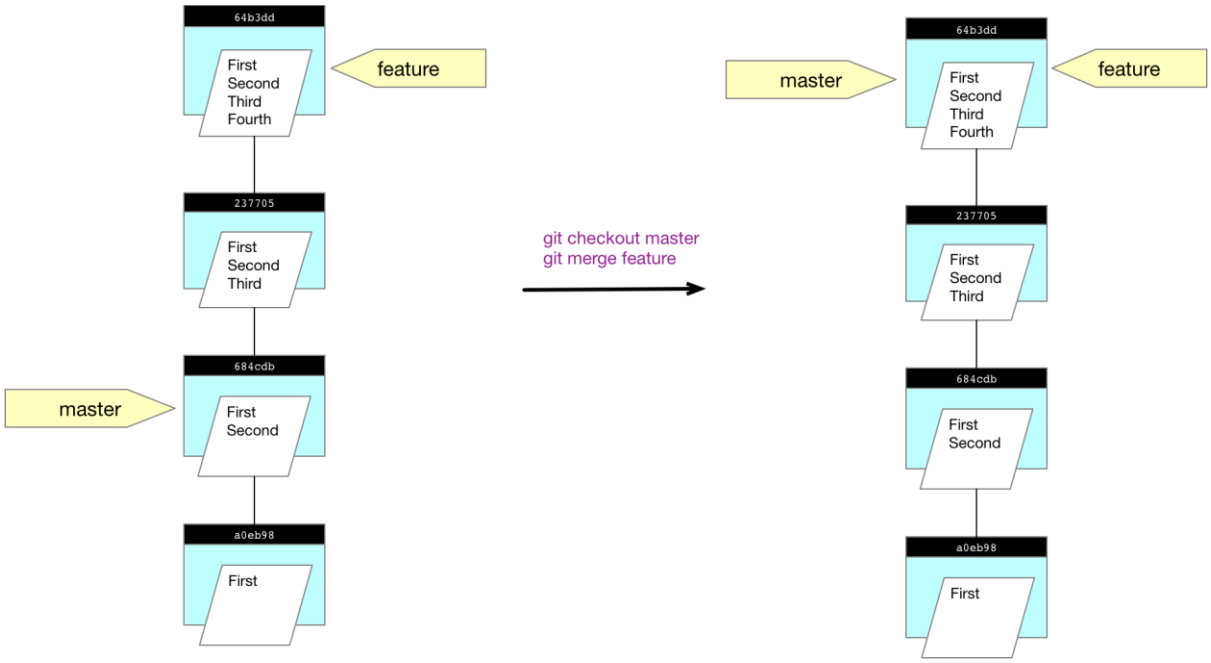
Figure 8 Rebase



Fast Forward Merge

When the source branch already has all the commit in the target branch merging is simply a case of move the target pointer to the same commit pointed to by the source branch. This is known as a fast forward merge

Figure 9 Fast Forward Merge



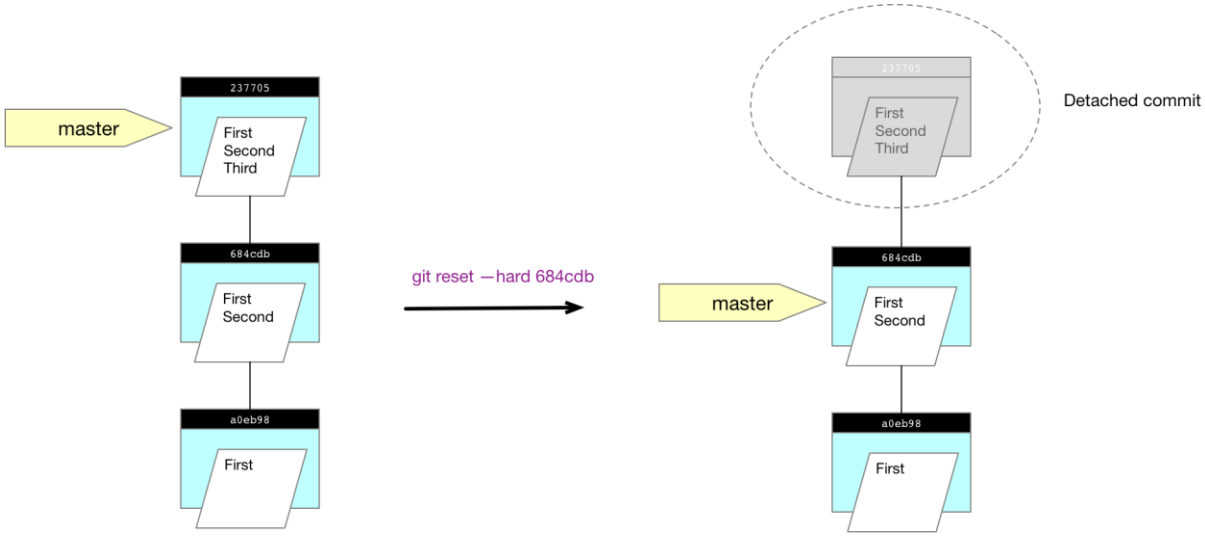
Rolling back changes

Reset

If we want to set the head back to a previous commit, we can use the reset command. Options control how many we go back too.

- Soft - only reset the head pointer. Leave the staging and working directory as is
- Mixed - set pointer and staging area. Leave working area as is
- Hard - update pointer, staging and working directory

Figure 10 Reset

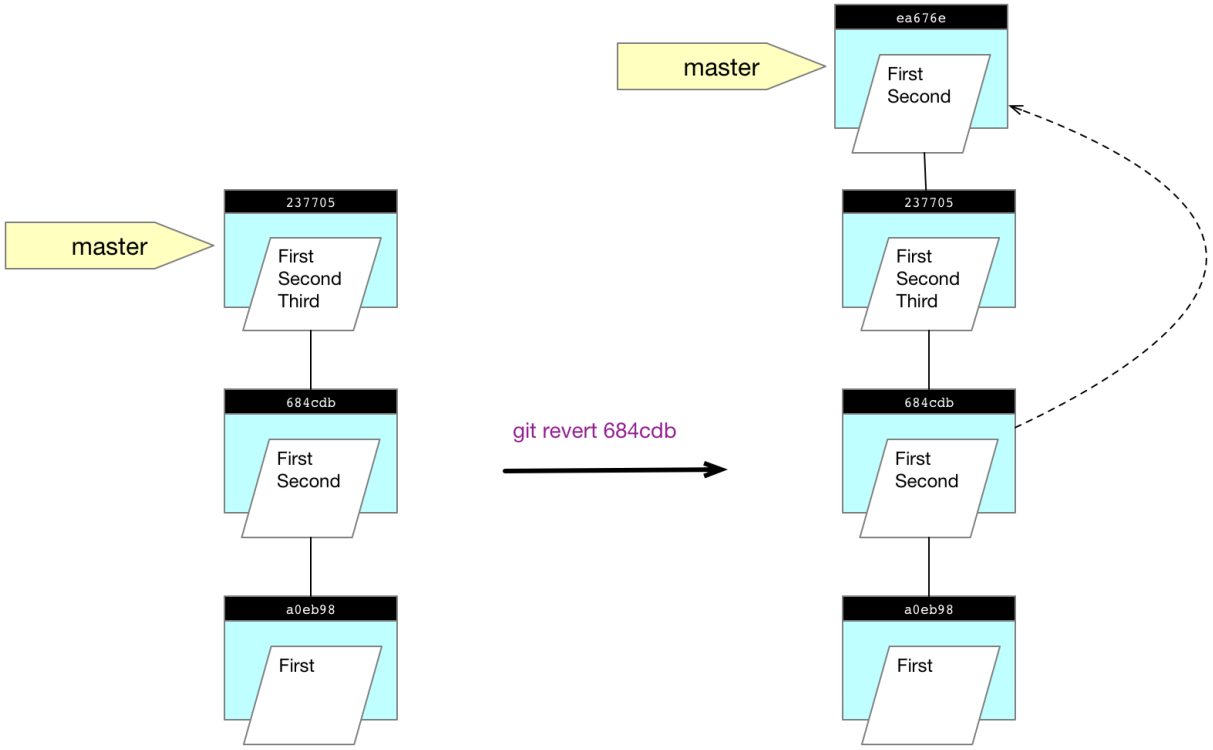


Command	Description
git reset --hard HEAD^	Reset head to previous commit
git reset -- hard HEAD~2	Reset head to previous commit
git reset HEAD --hard	Reset staging and working directory to latest commit

Revert

Unlike reset, revert effectively undo changes by taking the old code and reapplying it on top of the current code.

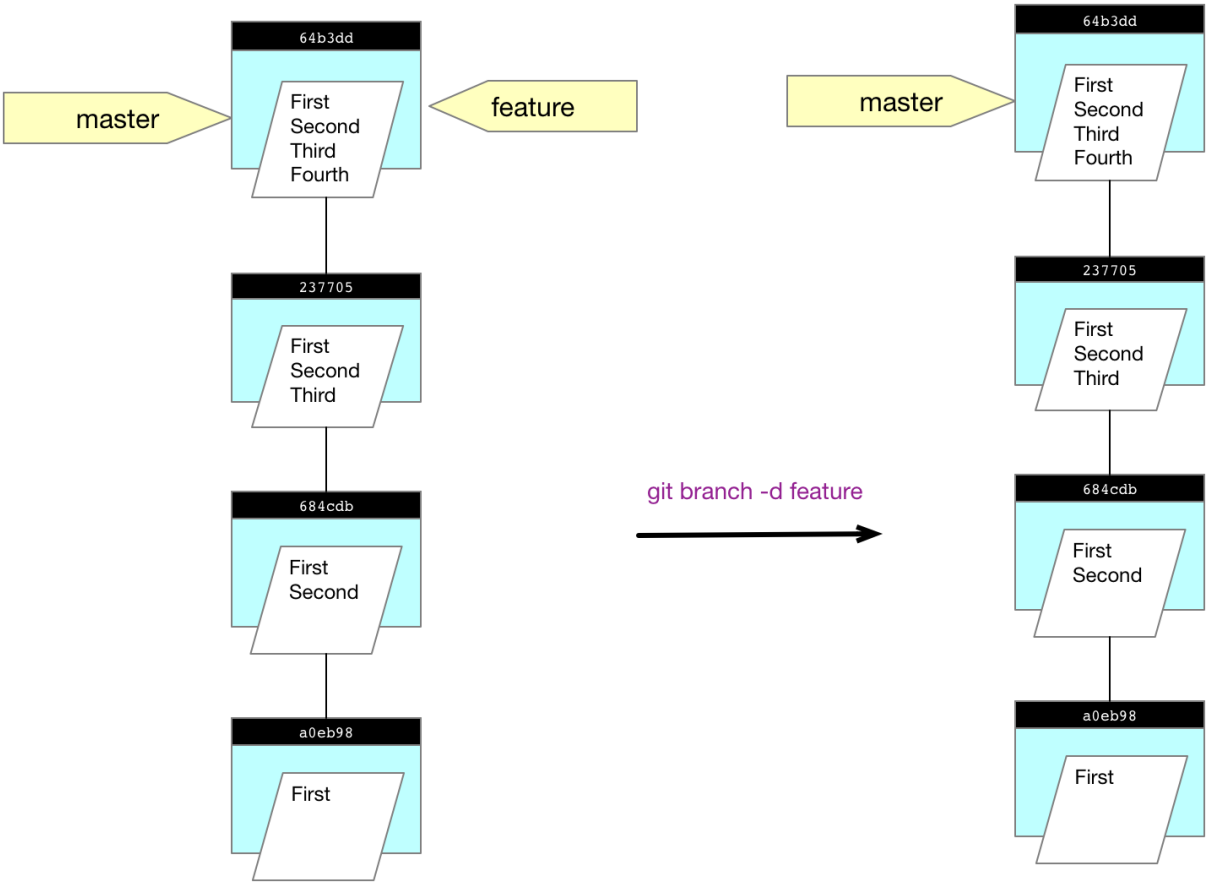
Figure 11 Revert



Cleaning Up

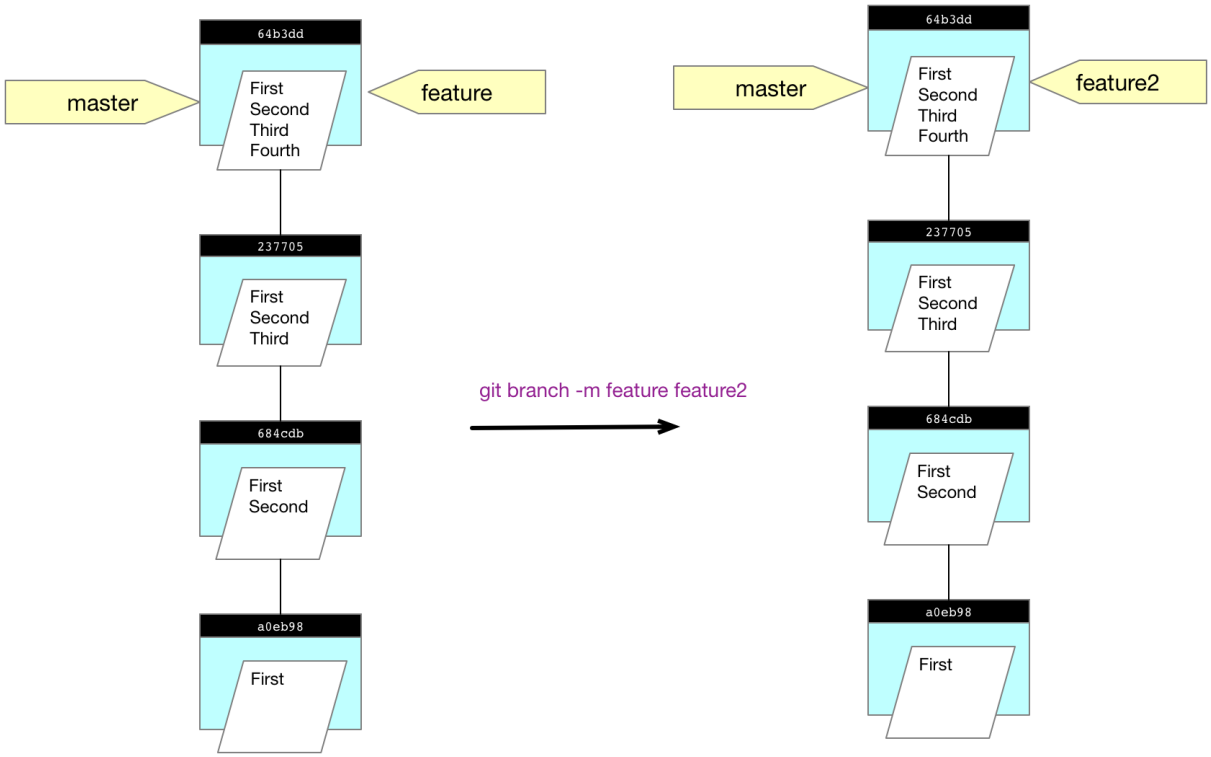
Delete Branch

Figure 12 Delete Branch



Rename branch

Figure 13 Rename Branch



Change commit messages

The following procedure shows how to change the commit messages on the last two commits. The first step is to run interactive rebase as followings

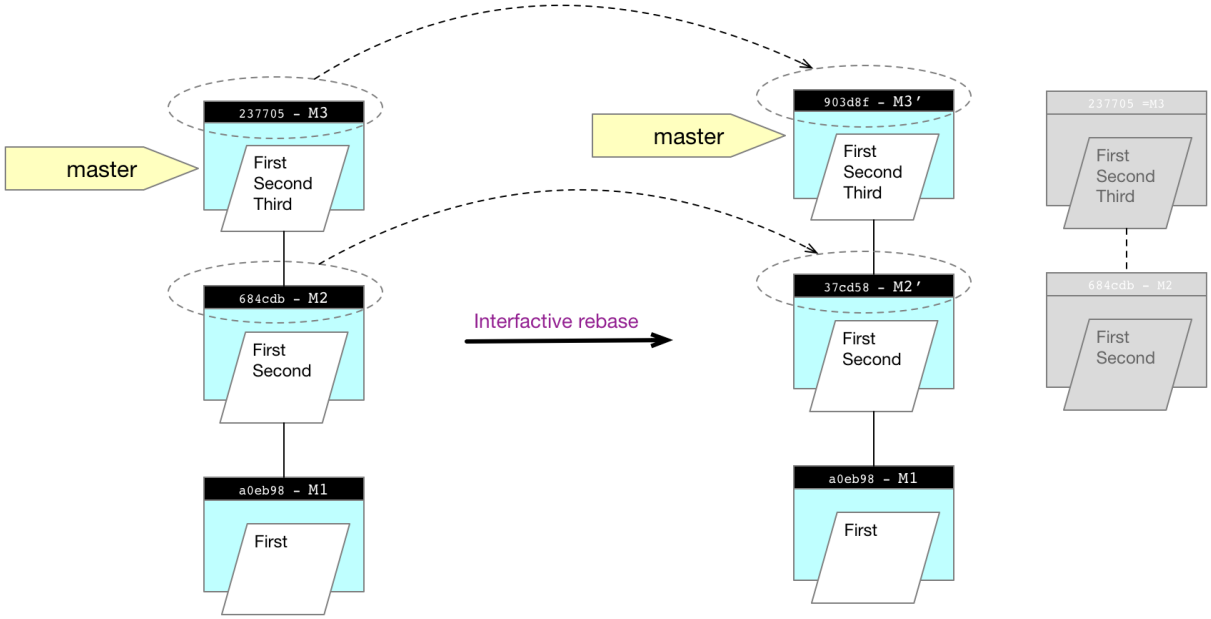
git rebase -i HEAD~2

An editor will pop up. Change the first two lines to look like this

reword 684cdb7 M2
reword 237705a M3

then an editor will open with the current commit message for each commit. Change it to be the new commit message in each case. For me I change M2 to be M2' and M3 to be M3'

Figure 14 Changing Commit Messages



Note that the original commits are not actually changed. What happens is two new commits are creates as clones of the original commits with their commit messages changed. The original commits are still in the repository, but they are now **detached**.

Combining commits

We can also use interactive rebase to combine commits together

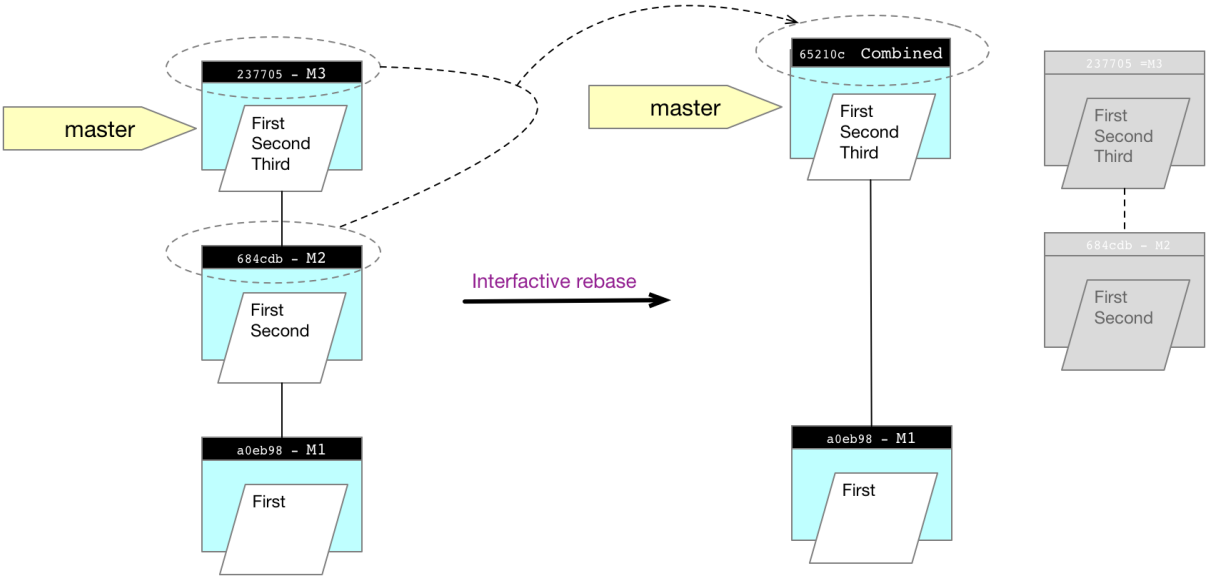
first step is to run interactive rebase as followings

git rebase -i HEAD~2

An editor will pop up. Change the first two lines to look like this

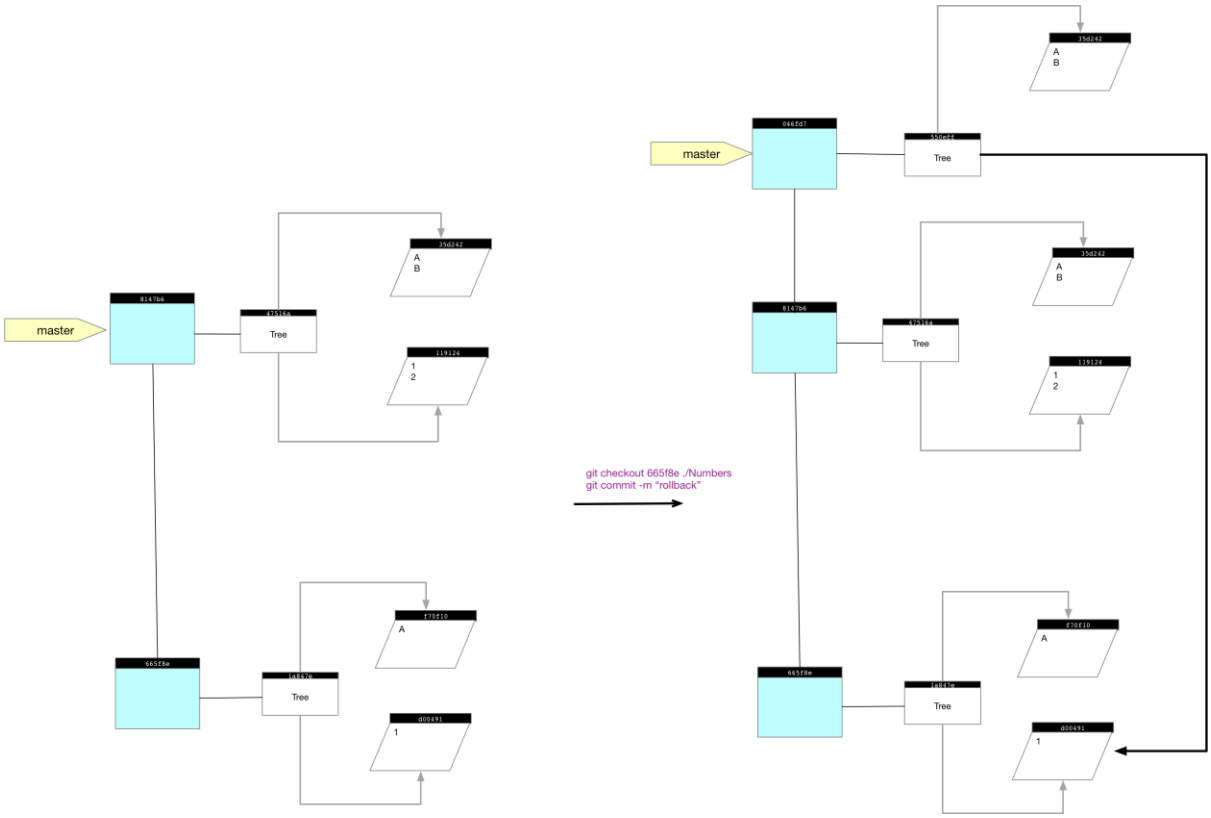
pick 684cdb7 M2
squash 237705a M3

Figure 15 Combing Commits



Rollback to old version of single file

Notice how we rollback to a previous version of a single file. First we run git checkout <SHA1> <FilePath> and then we check in a new commit whose tree points back to the old version



Commands

Table 3 Logging Commands

Command	Description
git log –oneline	Show one line per commit
git log --oneline -2	Show last two commits
git log --author "Kenneth Wilson"	Show commits by author
git log --since=2.weeks --until 12.31.2017	Show commits in date range
git log --oneline –nameonly	Show the files in the commits
git log --oneline --nameonly -- some/path	Show only the files in one path
git log --pretty=format:"%h (%ad %an) %s %d" --graph --date=short	Pretty print the log
git log -Smystring	Search for a string
`git config --global alias.hist 'log --pretty=format:"%h %ad | %s%d [%an]" --date=short'`	Set an alias for pretty print

Table 4 Diffing Commands

Command	Description
`git diff –staged`	Diff the staging area against the repository
`git diff HEAD`	Diff the working directory against the repository
`Git diff –name-only`	Show the files that have changed and not the changed
`git diff 1216756 811cbe4`	Diff two changelists

Table 5 Diffing Commands

Command	Description
`git branch mybranch`	Create a branch
`git checkout mybranch`	Switch head to new branch and bring its contents into working dir
`git checkout -b mybranch`	Creates branch and checkout in one step
`git branch`	List all branches
`git branch -v`	List branches plus info for last commit on each branch
`Git branch -d mybranch`	Delete the branch
`Git branch -m newname mybranch`	Rename the branch

Table 6 Commands for examining repo

Command	Description
git cat-file -t 37038bd	Show the type of the object with hash code starting 37038bd
$ git cat-file -p 37038bd	Show the contents of the object with hash code starting 37038bd
$ git show –stat –oneline 543678	Show the changes in a changelist

How To

Remove a File Permanently from Repo

Replace the PATH-TO-YOUR-FILE-WITH-SENSITIVE-DATA with your file

```
$ git filter-branch --force --index-filter \
  "git rm --cached --ignore-unmatch PATH-TO-YOUR-FILE-WITH-SENSITIVE-
DATA" \
  --prune-empty --tag-name-filter cat -- --all
```