What is git?

Git is a distributed version control system (DVCS) for managing source code.

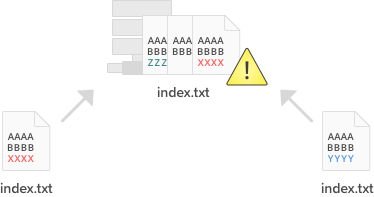
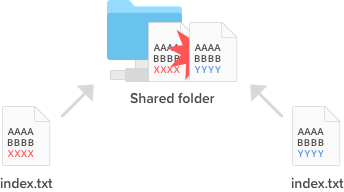
Ok, so what is version control? Simply put, version control is a system for tracking changes to files. As you modify files, the version control system records and saves each change. This allows you to restore a previous version of your code at any time.

A version control system like Git makes it easy to:

* Keep track of code history
* Collaborate on code as a team
* See who made which changes
* Deploy to staging or production

The **working tree**, or **working** directory, consists of files that you are currently **working** on. You can think of a **working tree** as a file system where you can view and modify files. The index, or staging area, is where commits are prepared. The index compares the files in the **working tree** to the files in the repo.

A **git** repository can support multiple **working trees**, allowing you to check out more than one branch at a time. With **git** worktree add a new **working tree** is associated with the repository. This new **working tree** is called a "linked **working tree**" as opposed to the "main **working tree**" prepared by "**git** init" or "**git** clone".

With Git, you can easily follow your source code's revision history and track changes. You can also go back in time to learn about how the version has changed and who has made the changes. When the latest version of a file is on a shared repository, Git will prevent unintentional overwrites by anyone on your team who has an older version of the file.Before a version control system vs.

**Git workflow**

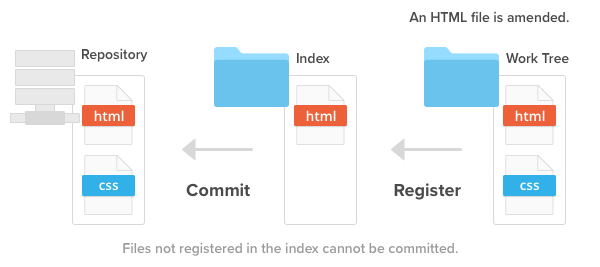
There are three main components of a Git project:

* Repository
* Working tree
* Index

The **repository**, or repo, is the “container” that tracks the changes to your project files. It holds all the commits—a snapshot of all your files at a point in time—that have been made. You can access the commit history with the Git log.

The **working tree**, or working directory, consists of files that you are currently working on. You can think of a working tree as a file system where you can view and modify files.

The **index**, or staging area, is where commits are prepared. The index compares the files in the working tree to the files in the repo. When you make a change in the working tree, the index marks the file as modified before it is committed.

The three main components of a Git project: the repository, index, and working tree.

Below is the basic Git workflow:

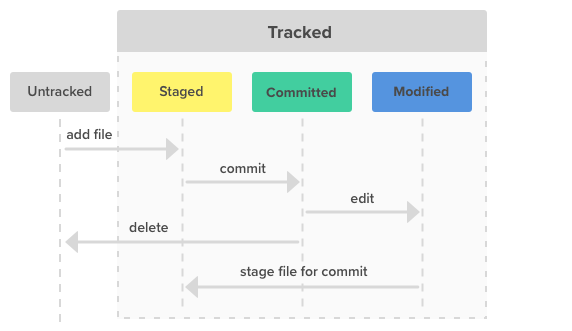
1. Modify files in the working tree.
2. Stage the changes you want to be included in the next commit.
3. Commit changes. Committing will take the files from the index and store them as a snapshot in the repository.

## Three states of Git files

As you can probably guess from the Git workflow, files can be in one of three states:

* Modified
* Staged
* Committed

When a file is first modified, the change can only be found in the working tree. You must stage the changes you want to be included in your next commit. The index contains all file changes that will be committed. Once you have finished staging files, commit them with a message describing what you changed. The modified files are now safely stored in the repo.

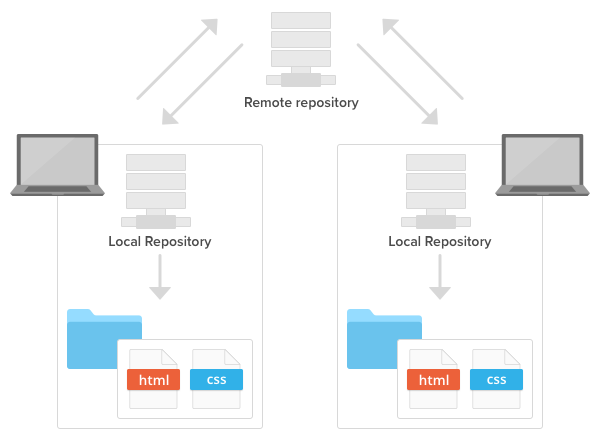
The three file states for Git: modified, staged, and commited

## Remote repositories and local repositories

There are two types of Git repositories: remote and local.

* A remote repository is hosted on a remote, or off-site, server that is shared among multiple team members.
* A local repository is hosted on a local machine for an individual user.

While you can take advantage of Git version control features with a local repository, collaboration features—like pulling and pushing code changes with teammates—will be better suited on a remote repository.

Git remote and local repositories working in harmony.

## How to create a repository

There are two ways to create a local repository on your machine. You can create a new repository from scratch using a file folder on your computer or clone an existing repository.

### Git init

You can create a new repo from scratch using the **git init** command. It can be used to introduce Git into an existing, unversioned project in order to start tracking changes.

### Git clone

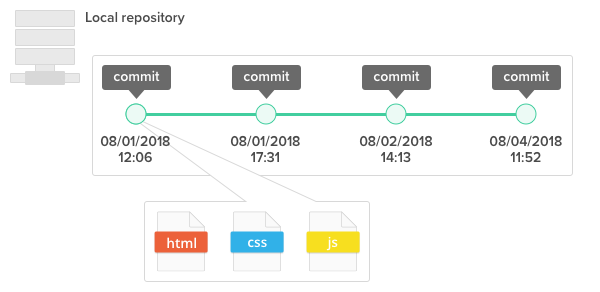
Use the **git clone** command to copy a remote repository onto your local machine.

By default, **git clone** automatically sets up a local master branch that tracks the remote master branch it was cloned from.

## Git commit

The **git commit** command enables you to record file changes in the repository's Git history.

By committing, you will be able to view all changes chronologically in the respective file or directory.

The commit history is stored in the local repository.

A 40-character checksum hash is used to uniquely identify a commit. You can use this checksum hash to retrieve the status or changes of files and directories that were made on the given commit in your repository.

# Undoing changes

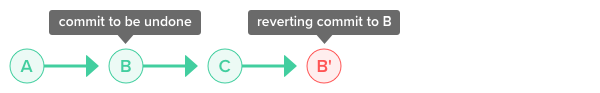
One of the most valuable features of Git is the ability to undo mistakes. When you make a new commit, Git stores a snapshot of your project so that you can go back to an earlier version when you need to.

There are two ways to undo changes: **git revert** and **git reset**.

## Git revert

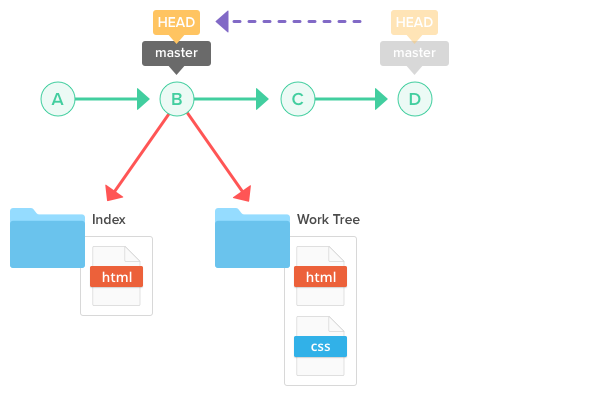
You can use the **git revert** command to safely undo a commit that has already been pushed.

While you can also delete a previous commit from the history using **git reset**or **git rebase -i**, it is generally not a good idea because it causes the remote repository to become desynchronized with the local repositories of other members.

git revert is the safest method of undoing changes.

## Git reset

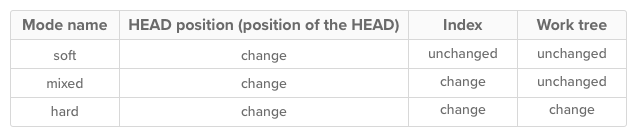
You can discard commits that you no longer need using the git reset command. You can specify the scope for the reset command by going into reset mode.

Use git reset to remove unnecessary commits.

There are three primary reset modes:

* Mixed (default)
* Soft
* Hard

Mixed mode restores the state of a changed index. Soft mode undoes a previous commit. Hard mode removes all traces of a commit. Below is a breakdown of each reset mode.

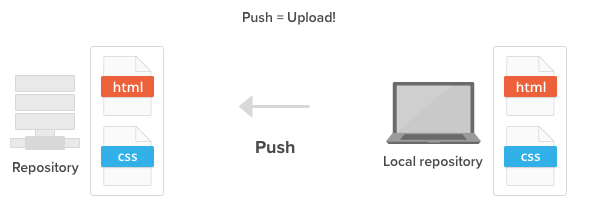
There are three reset modes: soft, mixed, and hard.

# Syncing repositories

Remote repositories allow us to share our changes with other members of the team. They can be on a private server, on a different computer than yours, or hosted using a service like Backlog. Wherever yours is hosted, you'll need to be able to sync your local repository with the remote repository frequently. You'll do this using three commands: git push, git pull, and git merge.

## Git push

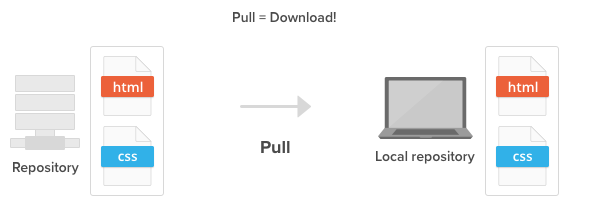
In order to start sharing changes with others, you have to push them to a remote repository using the "push" command. This will cause the remote repository to update and synchronize with your local repository.

Push your local changes to a remote repository.

## Git pull

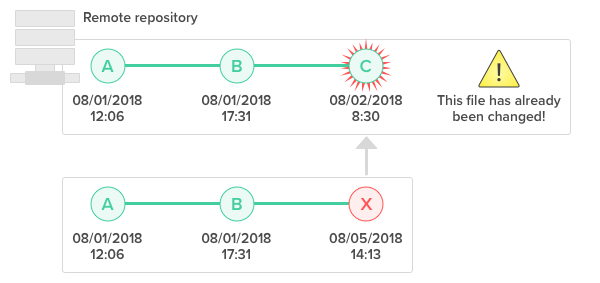
Whenever somebody pushes their changes to the shared remote repository, your local repository becomes out of date. To re-synchronize your local repository with the newly updated remote repository, simply run the git pull operation.

When the pull is executed, the latest revision history will download from the remote repository and import to your local repository.

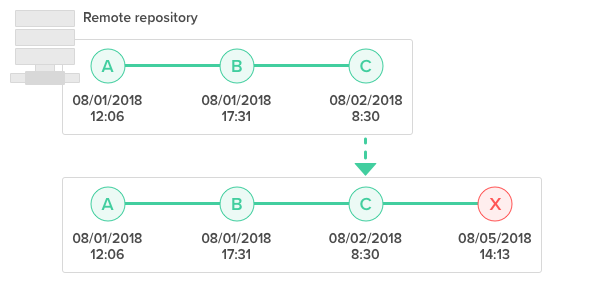
Pull changes from a remote repository to your local repository.

## Git merge

Your push to the remote repository will be rejected if your local repository is out of date, possibly because there are some updates on the remote repository that you do not have locally yet.

You are unable to push to the remote repository if your local repo is out of date.

If that is the case, you'll have to use the git merge command to grab the latest change from the remote repository before you are allowed to push. Git enforces this to ensure that changes made by other members get retained in the history.

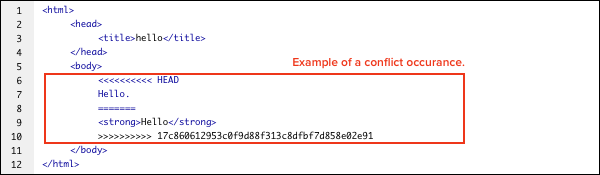
You must merge the latest changes before pushing.

During a "merge", Git will attempt to automatically apply those history changes and merge them with the current branch. However, if there is a conflict in changes, Git will throw an error prompting you to resolve the conflict manually.

## Resolve merge conflicts

When merging two branches, you may come across a conflict that needs resolving before you can properly complete the merge. For example, when two or more members make changes on the same part of a file in the two different branches (remote and local branches in this case), Git will not be able to automatically merge them.

When this happens, Git will add some standard conflict-resolution markers to the conflicting file. The markers help you figure out which sections of the file need to be manually resolved.

Example of a conflict occurrence.

In our example, everything above "=====" is your local content, and everything below it comes from the remote branch.

You must resolve the conflicting parts as shown below before you can proceed with creating a merge commit.

Revise the commit to eliminate the conflict.

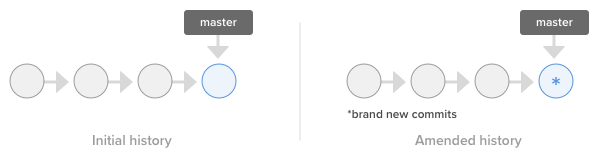
# Rewriting history

* Git commmit --amend
* Git rebase
* Git cherry pick
* Git merge --squash

There are times when you need to revise your local commit history. This can include anything from changing your commit message to changing the order of your commits to squashing commits together. In this section, we'll discuss how to rewrite history before sharing your work with others.

## Git commit --amend

You can modify the most recent commit in the same branch by running **git commit --amend**. This command is convenient for adding new or updated files to the previous commit. It is also a simple way to edit or add comments to the previous commit.

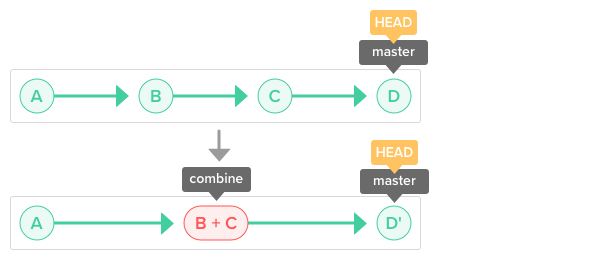
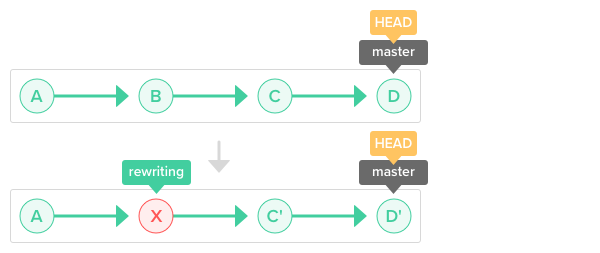
Use git commit --amend to modify the most recent commit.

## Git rebase

Rebasing is the process of taking all the changes that were committed on one branch and applying them to a new branch.

Run **git rebase** and add in the -i option to rewrite, replace, delete, and merge individual commits in the history. You can also:

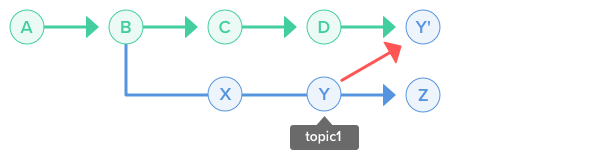
* Rewrite a past commit message
* Squash a group of commits together
* Add files that have not been committed

Identify the commit you want to rewrite and run the git rebase -i command.

## Git cherry pick

You can apply an existing commit from another branch to the current branch within the repository by using the git cherry-pick command. Cherry-picking allows you to:

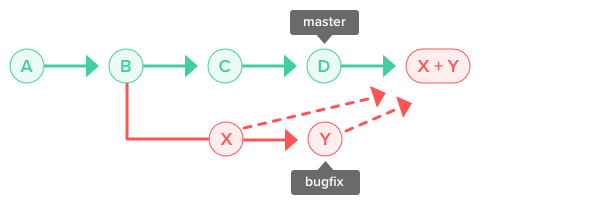
* Move a commit that was committed to the wrong branch to the right branch.
* Add a commit to the current branch based on an existing commit from another branch.

Use git cherry-pick to change the branch of a commit.

## Git merge --squash

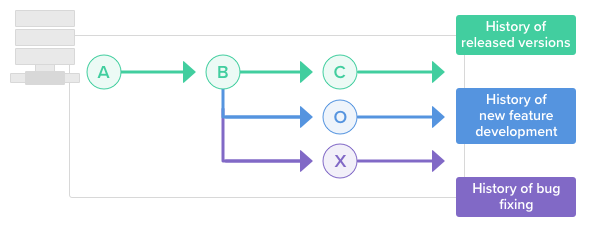
Squashing is the process of merging multiple commits into a single commit.

If you run **git merge** and the **--squash** option, a new commit will group all of the commits from that branch together. The commit will be used to merge into the current branch.

Use git merge --squash to unifying commits from a feature/topic branch into a single commit to be merged into your current branch.

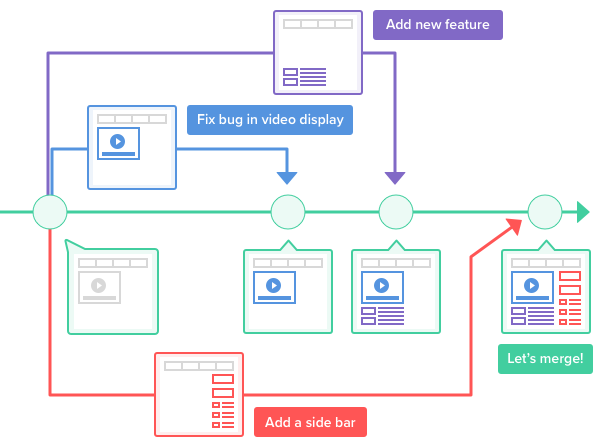
## What is a Git branch?

A Git branch is essentially an independent line of development. You can take advantage of branching when working on new features or bug fixes because it isolates your work from that of other team members.

A git branch is an independent line of development taken from the same source code.

Different branches can be merged into any one branch as long as they belong to the same repository.

The diagram below illustrates how development can take place in parallel using branches.

Multiple development projects taking place using the same source code.

Branching enables you to isolate your work from others. Changes in the primary branch or other branches will not affect your branch, unless you decide to pull the latest changes from those branches.

It is a common practice to create a new branch for each task (i.e., a branch for bug fixing, a branch for new features, etc.). This method allows others to easily identify what changes to expect and also makes backtracking simple.

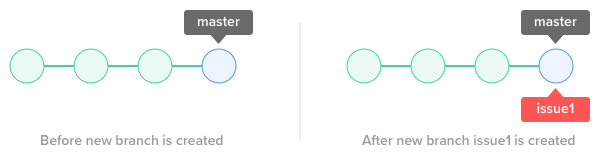
## Create a branch

Creating a new branch does not change the repository; it simply points out the commit

For example, let's create a branch called “issue1” using the command git branch.

git branch issue1

The illustration below provides a visual on what happens when the branch is created. The repository is the same, but a new pointer is added to the current commit.



## Git Stash

Whenever you switch to another branch with uncommitted changes (or new files added) in your working tree, these uncommitted changes will also be carried to the new branch that you switch to. Changes that you commit will be committed to the newly switched branch.

However, if Git finds a conflict between the files from the newly switched branch and the uncommitted changes from the previous branch, you will not be allowed to switch to the other branch. You must commit or stash those changes first before switching branches.

You can think of stash as a drawer to store uncommitted changes temporarily. Stashing allows you to put aside the “dirty” changes in your working tree and continue working on other things in a different branch on a clean slate.

Uncommitted changes that are stored in the stash can be taken out and applied to the original branch and other branches as well.

# Fetch remote branch

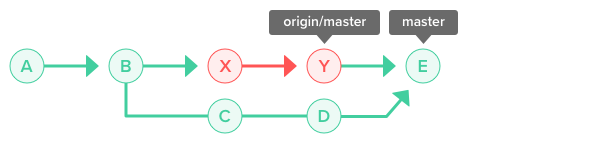
When you execute a pull, the changes from the remote branch automatically merge into your current local branch. If you want to obtain the remote changes but not have them merged into your current local branch, you can execute the **git fetch** command.

Fetch will download the changes from remote that do not yet exist on your local branch. The FETCH\_HEAD ref can be used to track the fetched changes from the remote repository.

The revision history will look like below when both the remote and local branch contain different descendants.

Fetch respective origins of both local repository and remote repository
  under a state where both repositories have commits advanced from BRevision history when remote and local branches have different masters.

Once changes are fetched, you can apply those changes to your local repository by merging in FETCH\_HEAD or by executing a pull.

After merging, changes will be applied to the local repo.

Once FETCH\_HEAD has been merged, the revision history will yield the same result as a **git pull** operation. Pull is essentially a simultaneous execution of fetch and merge operations.

**Integrating branches**

Once you are done working on a feature/topic branch (i.e., new feature or bug fix), you would typically merge it with a develop/integration branch. You can accomplish that by using the **git merge** or **git rebase** commands, although both commands will give you different results.

**Merge:** Retains all changes to and history of the merged branch. The revision history can become complicated after many merge commits.

**Rebase:** Maintains a clean revision history since merged commits are appended at the end of the target branch. Conflicts may occur more often than in the merge method, and they need to be resolved immediately.

You and your team should decide on which method of merging to use. If you want to keep your revision history simple, you can do the following :

* Use rebase on your feature/topic branch when you want to pull the latest change from the develop/integration branch.
* If you want to merge the change from your feature/topic branch to the develop/integration branch, rebase the feature/topic branch onto the develop/integration branch first. After which, merge the changes from the feature/topic branch into the develop/integration branch. This will be a fast-forward merge with no extra merge commits being created.

**Tags**

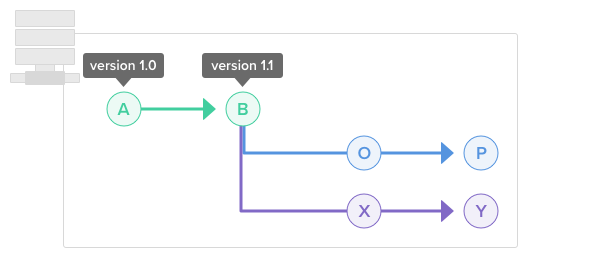
A Git tag is used to label and mark a specific commit in the history. Tags are commonly used to indicate release versions, with the release name (i.e., v1.0) being the name of the tag.

There are two types of Git tags:

* Lightweight tag
* Annotated tag

A **lightweight tag** is similar to a branch that does not change. It just points directly to a specific commit in the history. Lightweight tags are mainly used temporarily in your local workspace.

An **annotated tag** is checksummed and often used when you are planning to mark an important commit. You can add comments, a signature, the date, plus the tagger's name and e-mail.

Git tags in the master branch

|  |  |
| --- | --- |
|  |  |
| git init |  |
| Git status |  |
| Git add . / git add –A / git add filename |  |
| Git reset |  |
| Git reset filename |  |
| Git commit –m “message” |  |
| Git log / git log commit |  |
| Git show commitcode |  |
| Git show –name-only commitcode |  |
| Git show –pretty=”” –name-only commitcode |  |
| Git rm filename |  |
|  |  |
| Git commit –m “message” |  |
| Git rm –r foldername |  |
| Git rm –cached filename |  |
| Git log –oneline |  |
| Git log –graph –oneline |  |
|  |  |
|  |  |
| Git remote add aliasname githutrepo |  |
| Git push –u repoalias localbranch |  |
| Git clone githubrepo |  |
| Git pull githubrepo / git pull githubrepo/alias branchname |  |
|  |  |
| Git branch  Git branch newbranchnane |  |
| Git checkout branchname |  |
| Git merge branchname |  |
| Git branch –d branchname |  |
| Git reset –hard HEAD~ | Switch into other branch and rebase master |
| Git checkout newbranch |  |
| Git rebase master |  |
| Git tag tagname | Git tab testtag |
| Git tag |  |
| Git log –decorate |  |
| Git tag –a tagname |  |
| Git tag –am “message” tagname |  |
| Git tag –n |  |
| Git tag –d tagname |  |
| Git revert HEAD [master d47bbld] Revert “comments” |  |
| git reset --hard HEAD~~ HEAD is now at 326fc9f |  |
| git reset --hard ORIG\_HEAD HEAD is now at 0d4a808 |  |
| git rebase -i HEAD~~ |  |
|  |  |

54

Just undo the last commit:

git reset --soft HEAD~

Or undo the time before last time commit:

git reset --soft HEAD~2

Or undo any previous commit:

git reset --soft <commitID>

(you can get the commitID using git reflog)

When you undo a previous commit, remember to clean the workplace with

git clean

git revert HEAD~3

Revert the changes specified by the fourth last commit in HEAD and create a new commit with the reverted changes.

git revert -n master~5..master~2

Revert the changes done by commits from the fifth last commit in master (included) to the third last commit in master (included), but do not create any commit with the reverted changes. The revert only modifies the working tree and the index.