Basics of Version Control with Git

The Art of Computer Programming

Pierre Senellart

PSL UNIVERSITÉ PARIS

12 January 2026

Outline

The Problem

- Programming produces many versions of the same files
- Typical ad-hoc solutions:
  - only keeping the latest version
  - maintaining manual versions by hand: `project_final`, `project_final2`, `project_final_really`
  - manual copies of directories to share with others
- These approaches fail when:
  - you want to go back in time
  - you work with others, with more or less independent contributions
  - you introduce a bug and don't know when
  - you want to set up some automated backup mechanism
- We talk about programming here, but the same problem arises for all kinds of computer-based creative tasks: writing reports, articles, theses; computer-aided design; illustration or animation tasks

What Is Version Control?

- A version control system (VCS):
  - records the evolution of files over time
  - keeps a structured history of changes
  - allows recovery of previous states
  - supports collaboration
- Every change is:
  - attributed to an author
  - timestamped
  - described by a message
- Version control is a fundamental tool for programmers

Version Control Vocabulary

- Repository
  - A collection of files and their full history
  - Can be local or hosted remotely (e.g., GitHub, GitLab)
- Commit
  - A recorded snapshot of the repository at a given time, a specific version
  - Identified by a unique identifier, associated with other metadata (message, author, etc.)
- Branch
  - A named line of development in the repository
  - Allows independent evolution of the code
- Clone
  - A local copy of a repository
  - Contains all files and the complete commit history

A Short History of Version Control

- Early version control systems (1980s–1990s):
  - RCS, CVS
  - centralized model
  - limited support for branching and merging
- Second generation:
  - Subversion (SVN)
  - still centralized, but more efficient, robust, flexible
- Modern systems:
  - distributed version control systems (DVCS)
  - Git, Mercurial

Centralized vs Distributed VCS

- Centralized VCS (CVS, SVN):
  - one central server
  - users get a working copy on their local machine, and inform the server of every change
  - history lives on the server
- Distributed VCS (Git, Mercurial):
  - every working copy includes the full history
  - commits are local
  - collaboration via exchanging commits
- Consequences:
  - offline work is possible
  - no single point of failure
  - branching (see later) is cheap

Outline

Why Git?

- Git is a distributed version control system:
  - every developer has a full copy of the history
  - no central point of failure
- Designed for:
  - speed
  - branching and merging
  - large projects
- Git is:
  - the de facto standard (> 90% market share)
  - used by large code hosting sites such as GitHub, GitLab, Bitbucket

The origin of Git

- Git was created in 2005 by Linus Torvalds
- Original motivation:
  - manage the Linux kernel source code
  - thousands of contributors
  - very high performance requirements
- Replacement for proprietary VCS used at the time (BitKeeper)
- Design constraints:
  - speed
  - scalability
  - support for massive parallel development
  - integrity of the history

Key Design Ideas Behind Git

- Git is built around a few strong principles:
  - commits are identified by a cryptographic hash, not by a version number
  - history is a graph of commits
  - branches are lightweight references to that graph
- Practical consequences:
  - creating a branch is cheap
  - merging different commits is a first-class operation
  - history integrity can be verified
- Many Git concepts make sense once this design is understood

Repositories and Working Directory

- A Git repository is a project tracked by Git
- It contains:
  - your files (working directory)
  - a hidden `.git/` directory
- A directory is not a Git repository by default

The Three Areas

Area	Meaning
Working directory	Your current files
Staging area	What changes will go into the next commit
Repository	The committed history

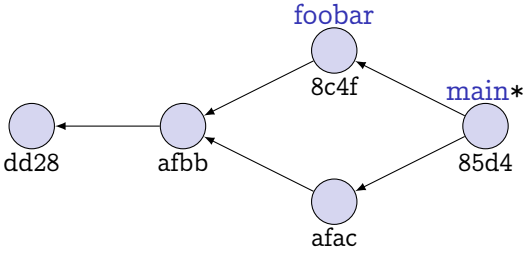
Commits

- A commit is a snapshot of the project (a specific version of all its files)
- Each commit has:
  - a unique hash, which serves as an identifier; 160-bit number (produced by the SHA-1 algorithm), written in hexadecimal, i.e., as a 40-hexadecimal digit number – often only the first few (e.g., 7) characters are used when no ambiguity
  - an author (name + email address) and precise timestamp
  - a descriptive message (describing this specific version, or more commonly the changes with respect to the previous version(s))
- Commits form a history directed acyclic graph (similar to a tree, except a commit can have multiple parents)

Branches and `HEAD`

- A branch is a movable pointer to a commit
- Always a default branch, usually called either main or master
- `HEAD` indicates on which commit you currently are
- Branches allow parallel work and experimentation

Example: A Repository and Its Commits



- Each node is a commit (a snapshot)
- Commits are identified by (to simplify) 4-digit hashes
- Arrows point to a commit's parent
- Branches (main, foobar) are indicated in blue
- `HEAD` is indicated by a *

Outline

Creating a Repository

Shell

```
git init
```

- Create a repository with no commits
- In practice will create a `.git` subdirectory to store the entire history

Checking Status

Shell

```
git status
```

- `git status` answers a simple question:
  "How does my working directory differ from HEAD?"
- It compares three conceptual states:
  - the last commit (the current snapshot)
  - the staging area (what will go into the next commit)
  - the working directory (your current files)
- Typical information reported:
  - modified files
  - newly created (untracked) files
  - staged changes ready to be committed

`git add`: Preparing the Next Commit

- `git add` selects changes to be included in the next commit
- It moves changes from the working directory to the staging area

Bash

```
git add report.py
```

- Effect:
  - the current content of `report.py` is copied into the staging area
  - future edits to `report.py` are not staged automatically
- Key idea:
  - `git add` does not create a commit
  - it defines *what the next commit will contain*

`git rm`: Recording File Deletions

- `git rm` removes a file and stages its deletion
- It keeps the repository history intact

Bash

```
git rm old_data.csv
```

- Effect:
  - the file is deleted from the working directory
  - the deletion is recorded in the staging area
- After the next commit:
  - the file disappears from future snapshots
  - it remains accessible in past commits
- Conceptually:
  - Git tracks changes to files, including deletions

git commit: Creating a Snapshot

- `git commit` creates a new commit from the staging area
- The commit becomes part of the repository history

Bash

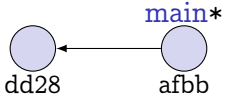
```
git commit -m "Fix sorting bug"
```

- Effect:
  - a new snapshot is created
  - it records exactly what was staged
  - the current branch now points to this new commit
- Important:
  - only staged changes are committed
  - unstaged changes remain in the working directory

Example: Creating Two Commits

Shell

```
git init
git add Matrix.cpp Matrix.h
git commit -m "Matrix class"
git add main.cpp
git commit -m "Main function"
```



Inspecting History

- Git stores the project history as a graph of commits

- `git log` lets you inspect this history

Shell
```
git log
```

- Shows:
  - commits in reverse chronological order
  - commit identifiers (hashes)
  - author, date, and commit message

Shell
```
git log --oneline --graph
```

- Compact, visual view:
  - one line per commit
  - graph of branches and merges

Seeing Differences

- Git can show differences between states

- `git diff` compares file contents line by line

Shell

```
git diff
```

- Shows:
  - differences between the working directory
  - and the staging area

Shell

```
git diff --staged
```

- Shows:
  - differences between the staging area
  - and the last commit

The .gitignore File

- Lists files Git should not track (compiled files, files produced by the editor, etc.) – anything that is not strictly useful
- Applies only to untracked files
- Files in the list will be ignored by `git status`, `git add`, etc.
- The `.gitignore` file itself must be added to a commit!

Text

```
*.o
a.out
__pycache__/
.vscode/
```

Outline

Undoing Safely

- Git provides reversible operations
- Prefer safe history-preserving commands

Restoring Files

Shell

```
git restore file.c
```

- Effect:
  - replaces `file.c` in the working directory
  - with the version from the staging area (i.e., the last staged/committed version)
- Use case:
  - abandon local edits that you do not want to keep

Shell

```
git restore --staged file.c
```

- Effect:
  - removes `file.c` from the staging area
  - keeps the working directory unchanged

Amending and Reverting

- Sometimes the history itself needs correction

Shell

```
git commit --amend
```

- Effect:
  - replaces the last commit
  - using the current staging area
- Typical uses:
  - fix a commit message
  - add forgotten changes to the last commit
- Important:
  - the commit graph changes (new commit hash)

Shell

```
git revert <commit>
```

- Effect:
  - creates a new commit
  - that undoes the changes of the given commit

Outline

Creating and Switching Branches

- A branch is a named pointer to a commit

Shell

```
git switch -c feature-x
```

- Effect:
  - creates a new branch named `feature-x`
  - makes it point to the current commit
  - switches the working directory to that branch

Shell

```
git switch main
```

- Effect:
  - moves the working directory to the `main` branch
  - updates files to match that branch's commit
- Key idea:
  - switching branches changes which snapshot you see

Merging

- Merging combines the histories of two branches

Shell

```
git merge feature-x
```

- Effect:
  - integrates changes from `feature-x`
  - into the current branch
- Conceptually:
  - Git finds a common ancestor
  - combines the changes since that point
- Result:
  - either a fast-forward
  - or a new merge commit with two parents
- Important:
  - merging preserves history

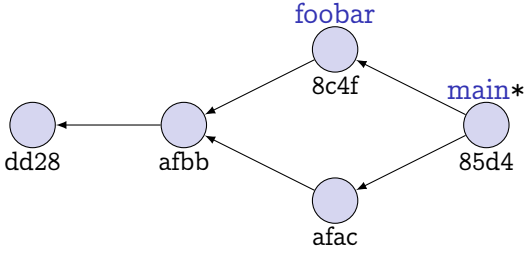
Example: A Repository and Its Commits

Shell

```
git switch -c foobar
git add main.cpp
git commit -m "Improve main function"

git switch main
git add .gitignore
git commit -m "Add a .gitignore file"

git merge foobar
```



Starting a Project: `git clone`

- In practice, work on a project usually starts by cloning a repository

Shell

```
git clone https://example.com/project.git
```

- Effect:
  - creates a new local directory
  - initializes a local repository
  - copies the complete commit history
- After cloning:
  - the remote repository is known as `origin`
  - a working directory is ready for use
- Conceptually:
  - cloning copies the commit graph, not just the files

Remotes

- A remote is another repository known to Git
- It usually represents:
  - a shared server (after cloning the repository from it)
  - or another developer's copy
- Remotes are identified by names:
  - most commonly `origin`
- Conceptually:
  - a remote is just another commit graph
  - Git keeps track of how your history relates to it

Push and Pull

- Collaboration consists in exchanging commits

Shell

```
git pull
```

- Effect:
  - fetches commits from a remote
  - integrates them into the current branch
- Conceptually:
  - equivalent to "download + merge"

Shell

```
git push
```

- Effect:
  - sends local commits to a remote repository
  - makes them visible to others
- Key invariant:
  - only commits are exchanged, never working files

Outline

Git in VS Code

- VS Code integrates Git
- Same concepts, graphical interface on top of Git
- Command-line interface (CLI) vs VS Code
  - CLI: universal, precise, scriptable
  - VS Code: convenient UI, excellent diff and merge tools
  - Recommendation: understand CLI, use VS as helper

Summary

- Version control is essential for programming
- Git records history and supports collaboration
- Learn the mental model, then the commands
- Practice on your own projects, and on `https://learngitbranching.js.org/`

Licensing