ECS 98F - Version Control (Using git)

Grant Gilson

Copyright © 2020-2021 Grant Gilson, Stephen Ott, Joël Porquet-Lupine, Aakash Prabhu, Noah Rose Ledesma
[CC BY-NC-SA 4.0 International License]

Git

The stupid content tracker

- Tool created by Linus Torvalds in 2005 to replace Bitkeeper
- Distributed version control manager

Main goals

Accountability

- Who wrote the code

Software branches

- Different software versions, ensure bug fixes are shared

Record and policy keeping

- History of the project is recorded
- Designate protocols to enforce good code guidelines

Setup

Installation

```
$ sudo apt install git
```

One-time configration

```
$ git config --global user.name "<FirstName LastName>"
$ git config --global user.email "<YourEmailHere>"
```

Setup

Clone from an existing repository

- After you or your partner created it on `github` first (or whichever git platform you prefer: e.g., `gitlab`, `bitbucket`)

```
$ git clone git@github.com:<nickname>/<project>.git
$ cd <project>
```

Or create a new repository

```
$ mkdir <project>
$ cd <project>
$ git init
```

Problem Scenario

Large group project

You are working on a very large group project with a lot of contributors.

- Keep track of who changed what in the codebase
- Efficienctly merge and distribute new code to the codebase
- The ability to try out new features without breaking the team's code
- The ability to turn back time when new code breaks old code

Tracking

A naive approach

Use shared google doc to keep track of:

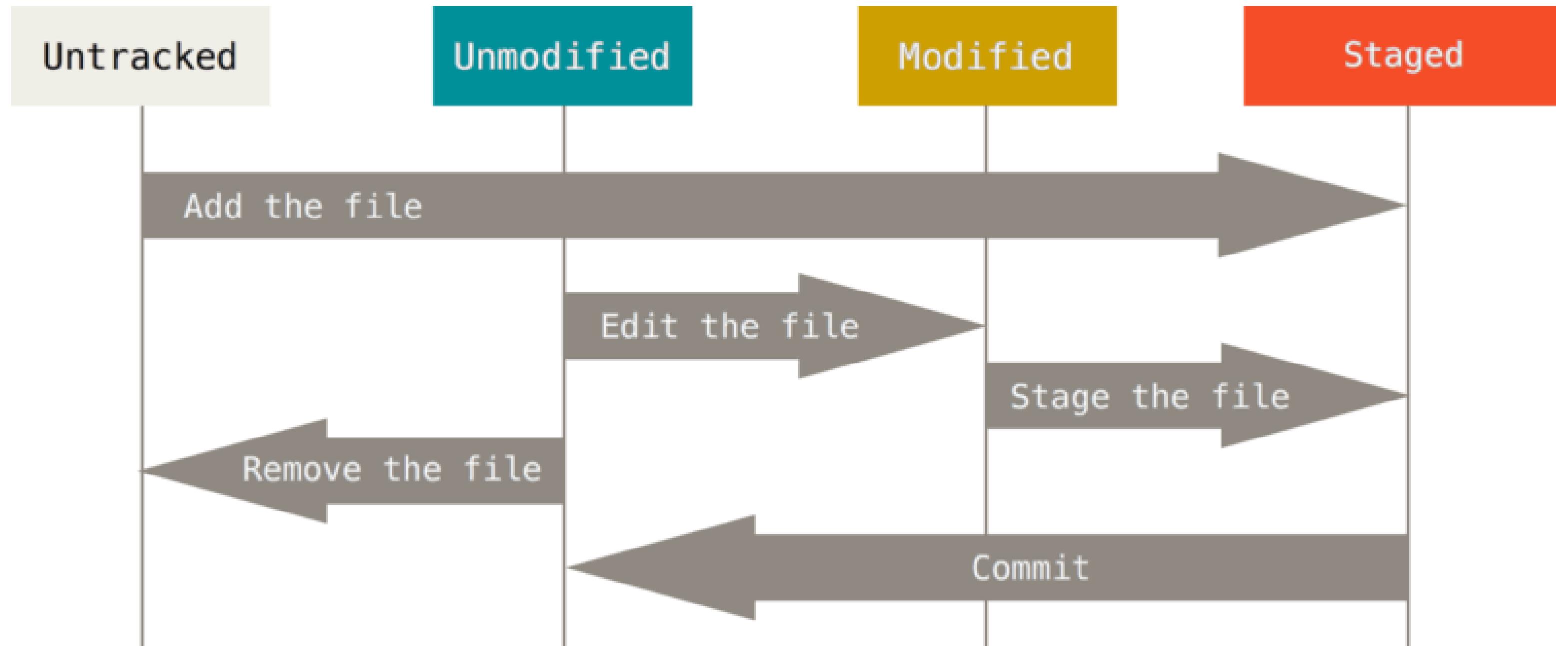
- What lines/functions changed
- Who changed them
- When they were changed
- Why they were changed

Problems

- Shared doc gets very large and unreadable
- Requires the author to remember every function/line they change
- Time spent not moving the project forward

Tracking

The cycle of git



Tracking

Checking file status

```
$ git status
```

- Returns status of each file in the repo

Tracking and staging

```
$ git add <filename>
$ git add <dirname>
```

- Adds the file to tracking if it was previously untrack
- Adds the file to the staging area

Committing the staging area

```
$ git commit # opens default editor
$ git commit -m "High level message of what I changed"
```

- Moves all staged files back to unmodified state
- History has been written to the repository

Tracking

Viewing history

To view changes at the commit level

```
$ git log
... commit_hash ...
... commit message ...
```

In order to see changes for a particular commit

```
$ git show 78d1470bea3581dd1f77aaede584c1f6a2ce491d
... commit message...
...lines added/removed...
```

Determine who last modified each line in a file

```
$ git blame <filename>
```

Distributing

A naive approach

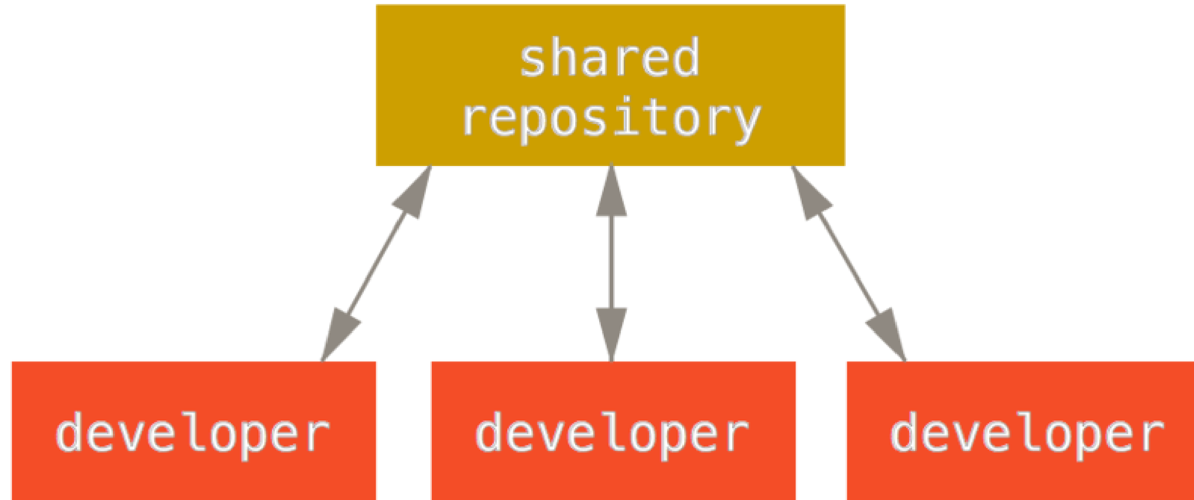
- Email your colleagues copies of your code
- You colleague `diff` their version with yours and manually applies patches

Problems

- Leaves the burden of merging in code to your colleages
- Which person's copy is the real version of the project?

Distributing

Centralized workflow



- *Real* Version is the shared repo
- Everyone syncs to the shared repo
- Everyone up-to-date can write to the shared repo
- Everyone gets a copy of the shared repo

Distributing

Publishing

```
$ git push <remote> <branchName>
$ git push origin main  # example
```

- Publishes local changes to shared repo

Updating

```
$ git pull <remote> <branchName>
$ git pull origin main  # example
```

- Fetches changes from shared repo
- Attempts to merge changes

Distributing

Dealing with conflicts

The merge process is not perfect, conflicts can arise.

```
$ git pull
...
Auto-merging test1.c
CONFLICT (content): Merge conflict in test1.c
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git status
...
  Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:  test1.c
...
```

Distributing

Dealing with conflicts

Use you editor to resolve the conflict:

- Accept remote changes
- Continue using your changes
- Combination of remote and yours

When conflicts are resolved they need to be committed

```
$ git add test.c
$ git commit
```

Then push the results of the merge to the team

```
$ git push <remoteName> <branchName>
```

Trying out new features

A naive approach

1. Implement new idea in place
    - Feature didn't pan out `ctrl-z` until back to the original state
2. Create copy of file you want to change as `file.bak`, `file_working.bak`...
    - Feature didn't pan out: restore to `file.bak`

Problems

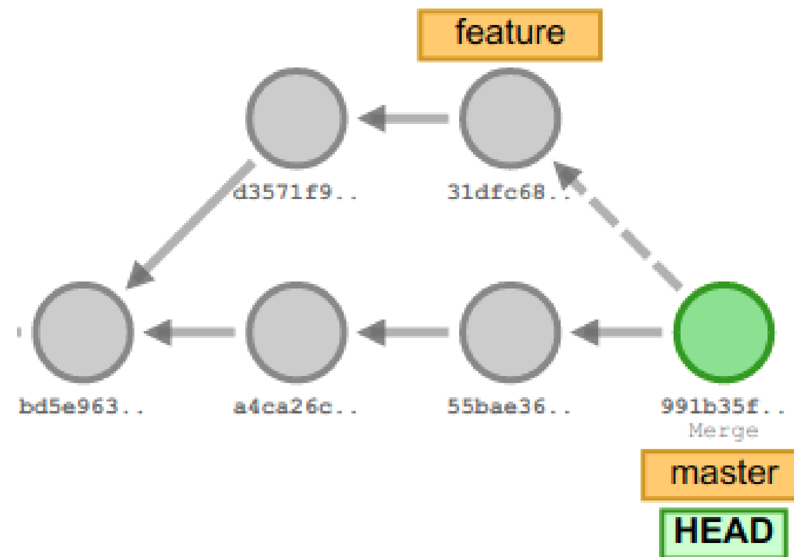
- Editor reloaded, can't `ctrl-z` anymore
- Easy to lose track of working versions of files

Trying out new features

Branching

Creates a isolated pesudo-new:

- working area
- staging area
- project history



- Commits in a branch are separate from the base branch until merged in
- Freely move between branches

Wrap up