Git, the "Undo" Button of Software Development

Step 1

"That feature was working last night, but then I tried to fix this other feature, and it broke everything! [insert curse word] I need to at least get it back to a working state!" –Every computer science student, at some point in their career

Whether we are talking about a piece of code for a programming assignment, or about your final essay after you made incoherent changes to it at 3:30 AM, or about your now unusable car after your friend Tim insisted he could install a new sound system because he's an electrical engineer and "a car is just a big circuit", we have all wished at some point in our lives that we could revert something to a previous unbroken state.

Luckily for us, there exists a solution to two of the three scenarios described above! **Version Control** is a system that records changes to a file or set of files over time so that you can recall specific versions later. We will be focusing on one specific version control system, **git**, which we will be using in this course to keep track of specific versions of our programming assignment code. Git may not be able to fix your car, but at least your grades will be able to flourish!

Step 2

There are three main types of version control systems (VCSs): **local**, **centralized**, and **distributed**.

A **local VCS** is essentially a more sophisticated way of saving different versions of a file locally (think "Final_Essay_rough.doc", "Final_Essay_final.doc", "Final_Essay_final_final.doc", "Final_Essay_final_seriously_this_time", etc.). In practice, a local VCS does not actually save all of these different versions of the same file. Instead, for each revision of your file, it keeps a "patch" (i.e., a list of edits) with respect to a previous version. Thus, to re-create what a file looked like at any point in time, the local VCF simply adds up all of the patches until that time point.

A local VCS solves the issue of keeping a personal set of revisions to a file, but as we know, most real-world projects are not one-person jobs: they typically consist of collaborative efforts of an entire team. A **centralized VCS** solves this need to have version control when dealing with revisions from multiple individuals. Centralized VCSs have a single server that contains all of the versioned files. Individual developers check out files from that central place when they want to make changes. This works quite well, but it has some flaws. One large flaw being: What if the centralized server goes down, even temporarily? During that downtime, nobody can collaborate at all, nor can they save any changes to anything they were working on. Also, if the server goes down permanently (e.g. the hard drive crashes), everything except for what the developers might have had checked out locally would be lost forever.

Consequently, we have the **distributed VCS**, such as Git. Distributed VCS clients don't just check out the latest snapshot of the files in the project: they fully mirror the repository locally. In other words, every clone is a full backup of all the data: if the centralized server dies, any local client repository can simply be copied back to the server to restore it.

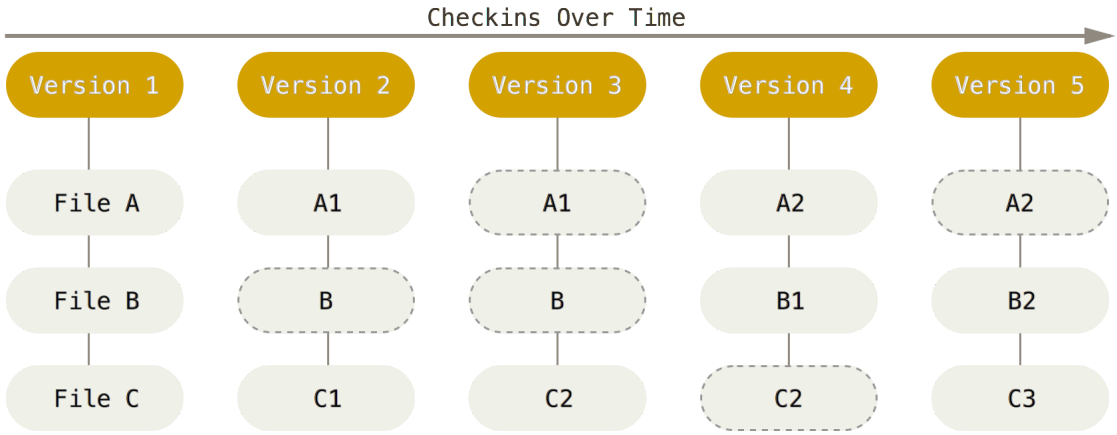
Step 3

EXERCISE BREAK: What type of Version Control System (VCS) is Git?

Step 4

Git is a relatively unique distributed VCS in that it treats versions as "snapshots" of the repository filesystem, not as a list of file-based changes. From the official Git website:

*"Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**."*



In general, the Git workflow is as follows:

1. Do a **clone** (clones the remote repository to your local machine)
2. Modify the files in your working directory
3. Stage the files (adds snapshots of them to your staging area)
4. Do a **commit** (takes the files as they are in the staging area and stores that snapshot permanently to your Git directory)
5. Do a **push** (upload the changes you made locally to the remote repository)

For more extensive information about Git, be sure to read the official Pro Git book.

Step 5

GIT PRACTICE: GitHub, a popular Git repository service, created an interactive Git tutorial that is of excellent quality. If you are not already comfortable with how to use Git, be sure to work through their tutorial, which has been embedded below:

Code School

1. *Initializing*
2. *Checking the Status*
3. *Adding & Committing*
4. *Adding Changes*
5. *Checking for Changes*
6. *Committing*
7. *Adding All Changes*
8. *Committing All Changes*
9. *History*
10. *Remote Repositories*
11. *Pushing Remotely*
12. *Pulling Remotely*
13. *Differences*
14. *Staged Differences*
15. *Staged Differences (cont'd)*
16. *Resetting the Stage*
17. *Undo*
18. *Branching Out*
19. *Switching Branches*
20. *Removing All The Things*
21. *Commiting Branch Changes*
22. *Switching Back to master*
23. *Preparing to Merge*
24. *Keeping Things Clean*
25. *The Final Push*

1.1 Got 15 minutes and want to learn Git?

Git allows groups of people to work on the same documents (often code) at the same time, and without stepping on each other's toes. It's a distributed version control system.

Our terminal prompt below is currently in a directory we decided to name "octobox". To initialize a Git repository here, type the following command:

```
git init
```

1. Console

1. Terms
2. Hints

Toggle Editor Mode

- Close
- Minimize
- Maximize

TryGit—834x310

```
Press enter to submit commands


>
```

Press Enter / Return to submit code in Console mode

- Close
- Minimize
- Maximize

My Octobox Repository

Advice

Directory: A folder used for storing multiple files.

Repository: A directory where Git has been initialized to start version controlling your files.

Step 6

In general, the extent of Git you will need to use for programming assignments in a data structures course or for a personal project will be as follows:

1. **Create** the Git repository

2. **Clone** the Git repository locally (`git clone <repo_url> <folder_name>`)
3. **Add** any files you need synchronized in the repository (`git add <files>`)
4. **Commit** any changes you make (`git commit -m <commit_message>`)
5. **Push** the changes to the repository (`git push`)

Step 7

Of course, this introduction to Git was extremely brief, but in reality, the bulk of your day-to-day Git usage will be pulling, adding, committing, and pushing. The more complex features are typically needed once in a blue moon, in which case you can simply Google the issue you're having and find an extensive StackOverflow post answering your exact question.

However, if you still feel as though you want more practice with Git, Codecademy has an excellent interactive self-paced free online course called Learn Git.