Version Control with Git

`Version Control Systems(VCS)` are software tools that help software teams manage changes to source code over time.
VCS allows programmers keep track of every modification to the code in a special kind of database, and turn back the clock and compare earlier versions of the code to help fix the mistake

`Git` is a distributed VCS, designed to track changes in source code, enabling multiple developers to work together on non-linear development.

- **Distributed Architecture**: Every developer's working copy of the code is also a repository that can contain the full history of all changes.
- **Branching and Merging**: Git offers excellent tools to branch and merge, making it possible for developers to have multiple local branches that can be entirely independent of each other.

`GitHub` is a cloud-based hosting service that lets you manage Git repositories.

- Pull Requests: An essential feature for team collaborations, which allows to notify others about changes you've pushed to a branch in a repository on GitHub.

Getting Started

1. Cloning a Repository (like GitHub)

```
git clone URL
```

2. Initializing a New Repository

```
git init
git remote add origin REMOTE_URL
# `origin` is a conventional name for your primary remote repository
# Replace <REMOTE-URL> with the URL you copied from GitHub

# To ensure that your local repository is connected to the remote
repository, run
git remote -v
# which will list the remote connections for your repository
```

3. change directory into your local cloning repository to further process

```
cd PATH
```

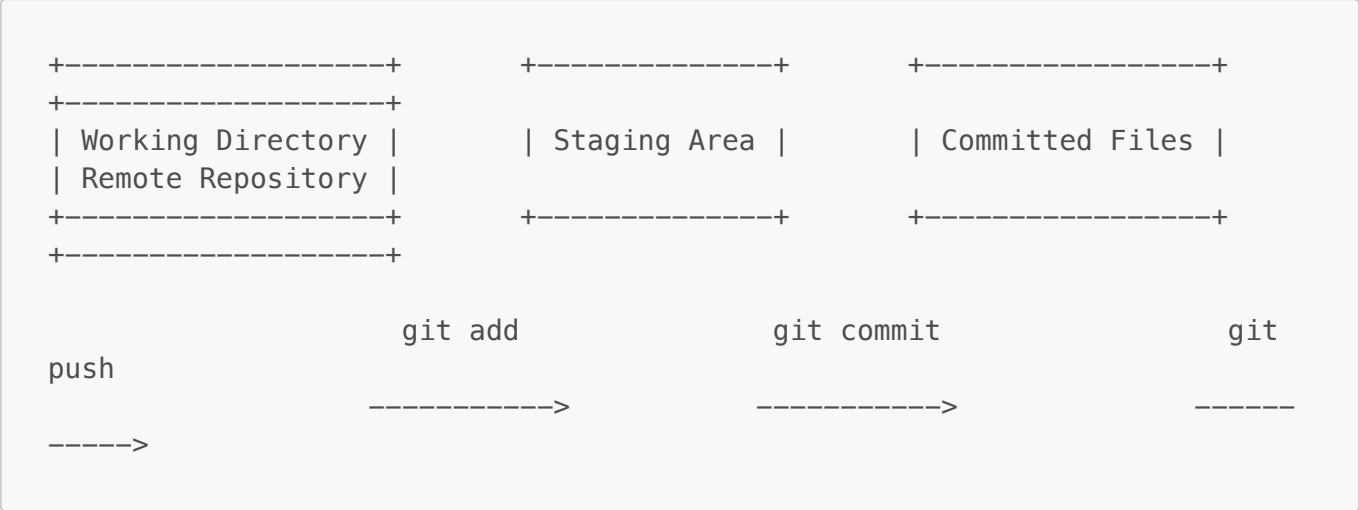
Cloning copies a repository AND creates corresponding working files.

`.git` subdirectory in the project, which makes the project directory a "repository". This directory is hidden by default.

NOTE: We don't necessarily have a "boss and servant" relationship between upstream and downstream repositories. Often times, a clone can become more active/popular than the original, in which case, the latter will start to sync with the former instead of vice versa.

Git Workflows

Git used `workflows` which can be broken into three steps and three states namely



Working Directory:

A `Working Directory` refers to the currently active directory on your filesystem that has been initialized with a Git repository. This is the directory where all your files and subdirectories that are **tracked** by Git reside.

The `Working Directory` is defined to be **CHANGED** whenever you modify files in your filesystem, regardless of whether those changes have been staged or committed.

`Modified` / `Untracked`

Modified files in your working directory.
Adding, Removing, and Updating any files inside the repository all belong to Modified state. In this state, Git knows the file has changed, but does not track it.

Untracked files in your working directory.
When a file is first created or brought into a directory that is part of a Git repository but has not yet been introduced to Git, it is untracked. Git does not monitor changes to untracked files until they are added to the tracking system.

- `$ git add file_name` - used to stage the file into commit (adds changes to the staging area)
- `$ git restore --stage file_name` - unstage the file from the commit

`Staged` & `Tracked`

Stage the changes, marking them for inclusion in the next commit
In order for Git to track a file, it needs to put(add) in the staged area. Once added, any modifications are tracked.

- `$ git commit` - commit the file inside the staged area (records the staged changes in the version history.)
    - `-m "MESSAGE"` (to attach a message about this commit)

Committed

Commit the changes, which takes the files as they are in the staging area and stores that snapshot permanently in your **LOCAL** Git directory.
Committing a file in Git is like a save point in many ways. Git will save the file, and have a snapshot of the current changes.

NOTE: Once you're satisfied with the changes in the staging area, you use the git commit command to take a snapshot of those changes. This snapshot is stored in your **local** repository. The commit has a unique identifier (a commit hash) and includes metadata such as the author, a message describing the changes, and so forth. At this point, the changes are safe on your **local** machine, but they're **NOT** yet shared with others or stored in a remote location.

After committed to your local machine, you need to use the following command to store it in a romote repository:

- `$ git push` - publish the local commits to a remote repository

Viewing Status

Checking the status of your repository:

```
git status
```

Each file can be in the following states:

- Staged
- Not staged but modified
- Untracked

Commit Messages

Commit messages are important because in essence, they help "market" your changes. They tell readers of the repository why certain commits were made and whether it was a "good" commit by explaining the *motivation* behind the changes. "Why are you making this change? Why shouldn't I just revert it?"

The rationale behind commit messages are similar to why you should comment your code. Oh yeah have I mentioned that:

> **COMMENTS SAVE LIVES. ALWAYS COMMENT YOUR CODE. PLEASE.**

There is overlap between comments in the source code and commit messages, but the primary distinction is the *audience*. Commit messages are more historically oriented, what you would tell the "software historian," people interested in the development of the repository as a whole. Comments in the source code are for the "current developer," people interested in having to study or change your code.

There are many style guidelines for commit messages out there, but here's Eggert's (which looks pretty standard in my experience).

Example commit message from the MIT repository shown in lecture:

```
Fix issues found by ASAN and Coverity

* tests/test_driver.pl: Preserve the LSAN_OPTIONS variable.
* tests/scripts/targets/ONESHELL: Don't set a local variable.
* tests/scripts/functions/let: Test empty let variable.
```

The first line should be at most 50 characters, and this acts as the "subject line" for the commit, like the elevator pitch. This should give any readers the *gist* of the commit.

The second line should be empty, separating the subject line from the body.

The remaining lines should be at most 50 lines, each at most 72 characters per line. Here you describe the finer details of the commit. You can use paragraphs, `*`-bulleted lists, etc.

Exploring the Log

Display information about every previous commit leading up to the current version, in reverse order by default:

```
git log
# displays the commit history of the entire repository

# the following is an output example
commit 9a2d3b4c5f6e7d8c9f0a1b2c3d4e5f6789012345 # Commit ID
Author: Jane Doe <jane.doe@example.com>
Date:   Wed Feb 15 14:03:12 2024 -0500

    Add login feature
```

Each commit has a unique `Commit ID`, which is a 40-character hash string representing the commit's contents. (also called Commit Hash)

Customizing output of `git log`:

```
git log --oneline
# Condenses each commit to a single line, showing a shortened commit ID
and the commit message
```

```
git log --source
# Shows the source ref (branch name) where the commit comes from, which
can be useful if you're searching through multiple branches like `git log
--source --all`

git log --stat
# Provides a summary of the changes included in each commit, showing the
number of additions and deletions in files

git log --author="Author_Name"
# Filters the log to show only commits made by a specific author

git log --pretty=format:"%h - %an, %ar: %s %cn %cd"
# Customizes the output format to show the short hash (%h), author name
(%an), author date relative (%ar), commit message (%s), committer name
(%cn) and the commit date (%cd-)

git log --oneline --decorate --graph --all
# Display the commit history as a decorated graph, including all branches
(each commit to a single line)

git log -S"function_name"
# Searches the commit history for commits that introduce or remove an
occurrence of such specific string
```

Usage of `--`

```
git log -- file_name
git log -- PATH
# `--` is used to tell Git that what follows is a path or file name, not
another option or a branch name

git log --oneline
# if the following is an option, it should connect to `--` without extra
space
```

Formatting the Log

```
git log --pretty=fuller
```

The fuller format shows that every commit has an **author** AND a **committer**, each with their own dates. Git distinguishes between these two contributors. This separation is routine for many larger projects. The author would be the person that writes the code, and the committer would be the overseer that reviews the and confirms the changes. These fields *establish responsibility* for changes, a major reason for using version control in the first place.

One interesting thing you might notice about repositories that have been in development for a long time: you might notice that commit timestamps go back even before Git existed. This is because the project may have migrated from other version control systems, such as RVS and CVS, and the date data is preserved when the history is copied.

Narrowing the Log

Narrowing the log in Git is to filter the commit history for more targeted information(or range).

You can use the special `..` syntax to specify a commit range. The following displays the history between commits with ref `A` (exclusive) and `B` (inclusive), so like `(A, B]`. This is like "show me everything that led up to B, but exclude everything that led up to A":

```
git log A..B
```

Shows all commits from the very beginning of the project up to and including B

```
git log B
```

ps. Both A and B are Commit ID.

You can also use **"version arithmetic"** to get references to commits based on aliases, like branch names, tag names, `HEAD`, etc. More on commit notation in Git Internals. The official documentation is also nice: https://git-scm.com/docs/gitrevisions.

`HEAD` is a special pointer refers to the current commit your working directory is on. It's a moving pointer that changes as you switch branches or make new commits.

`^` is Parent References syntax specifies the parent of a reference, so `HEAD^` or `HEAD~` or `HEAD~1` means the commit just before `HEAD`, `HEAD^^` or `HEAD~2` means the grandparent commit, etc.

Example about showing the most recent commit:

```
git log HEAD^..HEAD
# Shows everything that led up to HEAD, but exclude everything that led up
to HEAD^

git log HEAD^!
# Shorthands achieves the same result
# A way to specify a commit but exclude its parents
```

Working Files and Index

Technically a plumbing command, this is Git's version of `ls`, where it displays the current *working files*:

```
git ls-files
```

This helps us distinguish between general files in the directory and files that currently *matter* with respect to the repository. These are the files that are currently being **tracked** by Git.

```
rm $(git ls-files)  # lol!
```

Oh yeah, finding content within files is such a common pattern that `grep` is built into Git:

```
git grep waitpid
```

The **index** is what you have ready for the next commit, but have not committed yet (namely, in the staging area).

Viewing Differences

`git diff` is similar to the GNU `diff` command, and like a more detailed version of `git status`.

> **HISTORICALLY:** The algorithm is very complex and was developed by a professor at the University of Arizona who went on to work on the Human Genome Project.

> **Exit Code:**
>
> 1. `git diff` with an exit status of `0` means there is **NO** difference, with an exit status of `1` means there **IS** a difference.
> 2. In Unix-like operating system shells and Git commands, an exit status of `0` represents **True**, while a non-zero exit status like `1` typically represents **False**.

1. Viewing the difference between the *working directory* and the *staging area*: `Δ(working directory vs index)`:

   ```
   $ git diff
   ```

2. This views the difference between the *latest commit* and the *staging area*: `Δ(latest commit vs index)`:

   ```
   $ git diff --cached
   $ git diff --staged  # equivalent
   ```

3. And this is `Δ(working directory vs last commit)`

```
$ git diff HEAD
```

4. Compare the grandparent commit to the latest commit:

```
$ git diff HEAD^^..HEAD
```

5. More examples of viewing the difference between two commits:

```
# Typically with SHAs of the specific commits you want
$ git diff REF..REF

# But you can also abbreviate the hashes:
$ git diff 5c6cb30..53bf6bd
$ git diff 5c6c..54bf

# But this has a limit. This fails:
$ git diff 5c6..53b

# As usual you can use the HEAD ref to reference commits relative to
# the last commit:
$ git diff HEAD~..HEAD
$ git diff HEAD~4..HEAD
$ git diff HEAD^..HEAD
```

Making Changes

Typical workflow:

1. Edit the working files.
2. Run `git add FILES...` to add the specified file contents to the index (the **staging area**, the **cache**). You can keep editing files and add any new changes to the staging area with the same command.
3. Run one of the `git diff` commands to verify that the changes are what you want.
4. Run `git commit`, which takes your index, makes a new commit, and puts it into the object database with the auto-generated checksum. In effect, it changes the commit `HEAD` references.

You're probably familiar with `git commit -m MESSAGE` that every Git crash course teaches you. This is useful for one-liners, but the default `git commit` actually drops you into your configured editor and allows you to write longer commit messages with the subject line and body format detailed above.

There is also `git commit -m MESSAGE FILE`, where `FILE` contains the extended message. This is useful for automating messages in scripting. Example:

```
git commit -m 'Fix issues from previous patch' README.git
```

Removing all **untracked files**:

```
git clean
```

This is useful for removing files created as part of some build process. If you're not sure, you can run a "what if" with the `-n` option:

```
git clean -n
```

> The `--dry-run`/`-n` switch is common to a lot of Git (and Unix in general) commands. It's a good way to "preview" the effects of a potentially destructive command instead of running it blindly right away.

There's also the `-x` option which cleans files that will even be ignored:

```
git clean -nx  # you best see what that would do first lol
git clean -x
```

Configuring Git

Git configuration is handled through various files, including .gitignore, .git/config, and ~/.gitconfig, each serving different purposes.

`.gitignore` File

Files listed in `.gitignore` are those you typically don't want to push to remote repository and share with others, such as caches.

.gitignore file specifies intentionally untracked files that Git should ignore.
Files listed in .gitignore won't show up in untracked files lists and won't be added to your repository.

.gitignore is like a configuration file that instructs how users run Git. It's under Git's control i.e. it'll show up in `git ls-files`.

What files should be ignored?

Files that we do not want to put under version control. Obvious candidates include:

- Temporary files, such as log files, cache directories (_.log, _.cache) `\#*`
- Machine-Dependent Code, such as compiled binaries or object files (_.o, _.exe)
- Imported files (external packages or libraries that should not be part of your source control)
- Sensitive Information
    - Authentication credentials (passwords/keys/etc.)
    - Hashes of passwords? If it's intended for authentication, this would be just as bad as raw passwords, so ignore them too. Hashes enable **rainbow attacks** on the passwords where attackers try to crack the checksum algorithm.

`.git/config` File

.git/config is located in each Git repository and contains configuration options specific to that single repository. These options include remote repository addresses, branches configuration, and more.

You can view the current configuration in `.git/config` with:

```
git config -l
```

This outputs the information stored in the editable `.git/config` file, which is specific to the current repository. Cloning a repository also copies the configuration file.

CAUTION: Be careful when editing this file, syntax errors can cause Git to malfunction and stop working.

`.git/config` is NOT under version control because it determines how Git itself functions and because it would introduce the problem of recursion. `.gitignore` IS under version control because it's like a message from the developer and contains information about how to manage the project actually being version controlled. You also don't need to worry about what's in `.gitignore` to use Git itself.

`~/.gitconfig` File

For **Global Git Configuration** (like name, email address, preferred text editor, etc.) that applies to all of your projects, settings are stored in the ~/.gitconfig file.
These settings can be added or modified either by directly editing the file or using the following command:

```
git config --global KEY VALUE
```

Set up global username and email:

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

You can also use the global .gitconfig file to set preferences that make your workflow smoother and more efficient across all your projects.
Other cool things you can specify in this file:

You can edit directly inside the file like:

```
[core]
    # If you don't like being dropped into Vim by default,
    # This sets it to VS Code (you can also use Emacs, etc.)
    editor = code

[alias]
    # Abbreviations
```

```
    s = status
    co = checkout
    cm = commit -m
    # etc.
```

Or use `git config --global` command:

```
# Specify default editor
git config --global core.editor "code --wait"

# Define useful aliases
git config --global alias.s status
git config --global alias.co checkout
git config --global alias.cm "commit -m"
```

Branching with Git

Branching allows multiple development lines to proceed in parallel. This feature is a cornerstone of collaborative work, enabling different features or fixes to be developed simultaneously without interfering with each other.

Branch Operations

1. Creating a Branch

`$ git checkout -b branch_name` - creates a new branch and switches to it

- `-b` - stands for "branch", indicating to create a new branch

`$ git branch branch_name` - creates a new branch if it does not already exist, but will not switch to it.

2. Switching a Branch

`$ git checkout branch_name` - switches from your current branch to another existing branch

- `-f` - stands for "force", proceed with the checkout even if there are potential conflicts or changes

3. Listing All Branches

`$ git branch` - shows all local branches

- `-a` will show both local and remote branches

4. Deleting a Branch

`$ git branch -d branch_name` - deletes the specified branch

Examples and Use Cases

1. Feature Development
When starting work on a new feature, create a new branch off the main branch using `$ git checkout -b feature/new-feature`. This isolates your work from the main codebase until it's ready to be merged back.

`feature/new-feature` is the name of the branch you are creating. Using a prefix of `feature/` is a convention that helps in organizing branches. It is not mandatory but recommended in many workflows for clarity.

2. Bug Fixes
Similarly, for bug fixes, you might create a branch like `$ git checkout -b fix/login-bug`. This allows for targeted work on fixing the issue without affecting ongoing development in other areas.

1. Merging

Once your work on a branch is complete and tested, you'll want to merge it back into your main branch (or another parent branch). `git merge branch_name` merges the specified branch into your current branch, combining the histories of the two branches.

Merging is one way to bring together multiple lines of development. You integrate changes **from** one branch **to** another branch. More technically, it is when you create a commit from two or more parents commits, which may or may not be named branch tips.

Mechanism

Suppose you just cloned the original romote repository into your local machine, then you have the local commit history be like:

```
(origin/master, master)
         v
(C1)<--(C2)
```

Then you commit a snapshot to your local `master` branch (but not yet push it):

```
   (origin/master)
         v
(C1)<--(C2)<--(C3)
                ^
            (master)
```

Unfortunately, someone on your team has pushed new changes to the remote `origin master` branch before you push your local changes.
Then the remote commit history looks like:

```
          (origin/master)
                v
(C1)<--(C2)<--(C4)
```

Now you pull from the remote repository and have a new version of local commit history:

```
          (origin/master)
                v
(C1)<--(C2)<--(C4)<--(C5) < (master)
         |            |
         +----(C3)----+
```

p.s. C5 is a new merge commit created by Git to merge the changes from C4 into your local master branch, which includes your C3 work.

If you push your work right now, your local commit history will look like:

```
            (origin/master, master)
                      v
(C1)<--(C2)<--(C4)<--(C5)
         |            |
         +----(C3)----+
```

Once again, before you push your C5 local changes, someone on your team pushes their work to the remote repository:

```
               (origin/master)
                     v
(C1)<--(C2)<--(C4)<--(C6)
```

After pulling from the remote repository, you will get the new local commit histroy like:

```
               (origin/master)
                     v
(C1)<--(C2)<--(C4)<--(C6)<--(C7) < (master)
         |                   |
         +----(C3)<--(C5)----+
```

NOTES:

`master` is a local branch in your repository.

`origin/master` represents the master branch on the origin remote.

Now, supposes

```
                     this is the merge commit
                                v
()<--(A)<--()<--()<--(Y)<--(Merged)<-- ...
      |                        |
      +------()<----(X)<------+
```

Git finds the common ancestor A, of the parent commits X and Y. Then, it runs computation on all 3. It is as if Git runs:

```
diff3 X A Y > combined.diff  # "3-way diff"
```

This file describes changes to change the common ancestor A to *either* X or Y. Git then applies those changes and creates a *new* commit instance for it, the **merge commit**.

The command to merge a branch named BRANCH_NAME into the current branch:

```
git merge BRANCH_NAME
```

What this does is:

1. Compute 3-way merges.
2. Replace working files accordingly.

Merge Conflicts

More often than not, the changes *collide*, resulting in a **merge conflict** (Merge Conflicts occurs while two or more teammates modify the same region of source code, and commit together). If there's a collision, Git modifies the affected files with a "replica" of the collision with the special notation. Without a GUI, conflicts in their raw form actually look like:

```
<<<<<<<
A (Current Change)
=======
B (Incoming Change)
>>>>>>>
```

Git actually modifies the content of the conflicting file with this pattern, conflicting text separated by special barriers. The one with < brackets shows the **CURRENT** content, and the one with > brackets shows the **INCOMING** content. To resolve the conflict, you need to edit this block to only include one version of this content ("accept current change" or "accept incoming change"). You can also accept both changes. You

could also leave the file in this conflicted state with the barriers, but that's stupid practice because if you do this on a source file, it will almost definitely be a syntax error.

Resolving conflicts: usually you would edit the conflicting files in your editor of choice and regularly check for further instructions with `git status`. You can just conclude the merge with:

```
git merge --continue  # but not recommended
```

For example:

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

This means the version in `HEAD` (your `master` branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the `=======`), while the version in your `iss53` branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by entering your local file and replacing the entire block with this:

```
<div id="footer">please contact us at email.support@github.com</div>
```

This resolution has a little of each section, and the `<<<<<<<`, `=======`, and `>>>>>>>` lines have been completely removed. After you've resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved. Staging the file marks it as resolved in Git.

2. Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`.

`Rebasing` in git can help you organize your local commit history into one single line.

For example:

Assume your local commit history looks like:

```
  (origin/master)
        v
(C1)<--(C3)
  |
(C2) < (feature)
```

You are currently on the `feature` branch, and you now run the `rebase` command: `$ git rebase origin/master`, you will get a new **RE BASE** branch `(C2')`:

```
  (origin/master)
        v
(C1)<--(C3)
        |
      (C2') < (feature)
```

As you see, the `rebase` command helps you to change the base of your current local branch to the `master` branch. (From C1 to C3, where C3 is the origin/master branch)

After rebasing, you can now merge it to the `master` branch via `$ git merge feature`:

```
     (origin/master, feature)
               v
(C1)<--(C3)<--(C2')
```

Merging vs. Rebasing Pros/Cons

Merging	**Rebasing**
+ Only one commit is created per merge.	− A new commit is made for every commit you rebase.
+ Does not change existing commit history, so you're less likely to screw over others working on the same branch.	− Changes existing commit history. If you misuse this command, you could mess up an important branch like `main` for everyone else. See the Golden Rule of Rebasing.
+ Your steps can be fully retraced because you know when each merge was performed.	− It is difficult to see when a rebase actually occurred.
− If you have to merge often, these merge commits may pollute your history and make it harder to understand.	+ No unnecessary merge commits polluting your history, making it easier to understand.

Merging	Rebasing
➖ Your history will still have interweaving branches that may make navigation (`git log`, `git bisect`, etc.) harder.	➕ Keeps your history as linear as possible, making it easier to navigate with such commands.

Another downside is that you may repeatedly rebase if progress resumes on the `main` branch while review was still pending.

IMPORTANT: The fact that rebasing manually linearizes history is a point of philosophical concern. Some may see this as an upside because it makes it easier to review, but others may see rewriting history as a bad thing.

> **ASIDE:** Often times, project managers use Git history to see which programmers are better and who deserve bonuses. This is not always the best approach. The history does not tell you the full story. There is a very tenuous relation between number of commits and the value of a programmer to a project. A glaring example at this time of writing is the mass layoffs at Twitter where *genius* Elon Musk thought to fire people who have written the least lines of code.

> The *essence* of Git and any version control system is that you are editing *changes* to source code, not the source code itself. Understanding this is what sets a **software developer** apart from an ordinary "programmer".

> If you find yourself in an environment where people are lying on purpose about commits in order to improve their job prospects or something, you're in the wrong company. **- Dr. Eggert**

3. Stashing

Rebasing is less "formal" than merging. Changes are looser, but are recorded as a sequence of commit nonetheless. Stashing is even less formal than rebasing. Changes are only floating around in your repository, waiting to be reapplied.

The scenario looks something like:

1. You're working on the next change in your branch.
2. You want to switch to some other branch NOW. You *could* commit what you have right now, but that is bad practice because you're essentially committing junk. "You want your repository to be in good shape at all times."
3. Instead, you could save your changes in an external file:

```
git diff > mywork.diff # generates a diff file of your uncommitted
changes and redirects the output to a file named `mywork.diff`. This
file now contains all the changes you've made since the last commit

git checkout -f  # forcefully switches branches and discards any
changes in your working directory.
```

4. Checkout to the other branch and do work on it

```
git checkout main
```

5. Checkout back your original branch and patch it.

```
patch < mywork.diff -r1
```

Git actually provides a way to do this within Git itself, using the `stash` command. At step 3, you would do something like:

```
git stash push
```

This saves the state of your working files in some part of the index. When you want to retrieve this state, you can get it from the stash stack with:

```
git stash apply
```

Debugging with Git

Bisecting (Binary Search)

Suppose you have a linear piece of history where somewhere between a stable version and the most recent commit, something went wrong. You can think of this problem of finding the first faulty commit as partitioning the timeline into OK and NG ("not good") sections, hence *bisecting*.

The timeline is "sorted" in that if you think of OK=0 and NG=1, the history will always be such that all NGs follow OKs.

```
                    |
(v4.3)<--( )<--( )<---( )<--( )<--(main)
OK        OK    OK  |  NG    NG     NG
                    |
```

This then becomes a classic *binary search* problem, where we can identify the first NG commit in O(logN) time.

Starting a bisect in Git:

```
#                 NG   OK
git bisect start HEAD v4.3
```

Then we tell Git to run your check script on each commit and use the exit status to determine if the commit is OK or NG:

```
#              vvvvvvvvvv any shell command
git bisect run make check
```

In this case, we use a Makefile with a `check` target that defines some test cases for the program

Of course, this also introduces the problem that if your test cases are buggy, then you may get false alarms. If you know ahead of time that a commit, say `v3`, will produce unreliable test results, you can skip it with:

```
git bisect skip v3
```

Collaborative Best Practices

Branches:

- `master` or `main`: stable branch
- `develop`: for development
- Each team member may create their branches for individual features/bugs

Protect the `master` branch. DON'T force-push; it could destroy the commit history. Don't be that guy.

On GitHub, use **pull requests** to merge changes into other branches. Pull requests may undergo a **review**. You can also use **issues** to assign bugs or features to team members.

Avoid merging temporary files or code meant for debugging (like dummy data).