

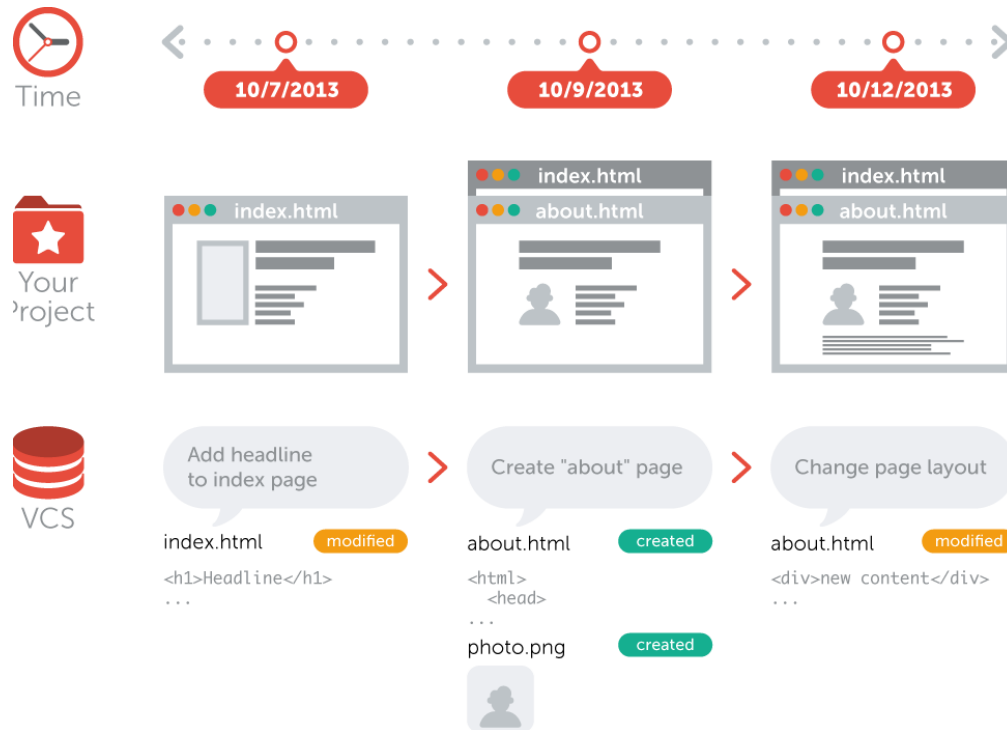
CS 398: Application Development

Using Git

Benefits; Git commands; Branching; Workflows.

Version Control w. Git

Version Control Systems (VCS) are software systems that track changes to your files. Examples: Git, Subversion (SVN), Perforce.

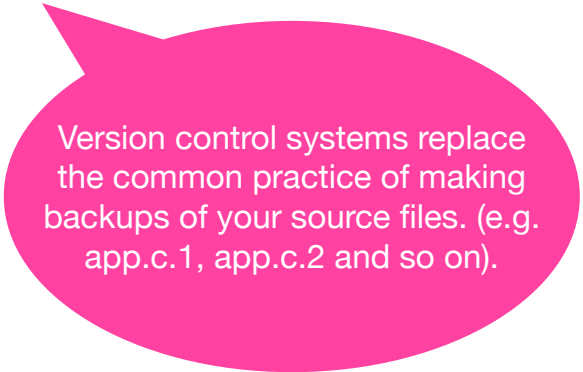


Git tracks changes to sets of files over time.

Why version control?

A VCS provides some major benefits:

- **History:** a VCS provides a long-term history of every file. This includes tracking when files were added, or deleted, and every change that you've made.
- **Versions:** the ability to version sets of files together. Did you break something? You can always unwind back to the "last good" change that was saved, or even compare your current code with the previously working version to identify an issue.
- **Collaboration:** a VCS provides the necessary capabilities for multiple people to work on the same code simultaneously, while keeping changes isolated.



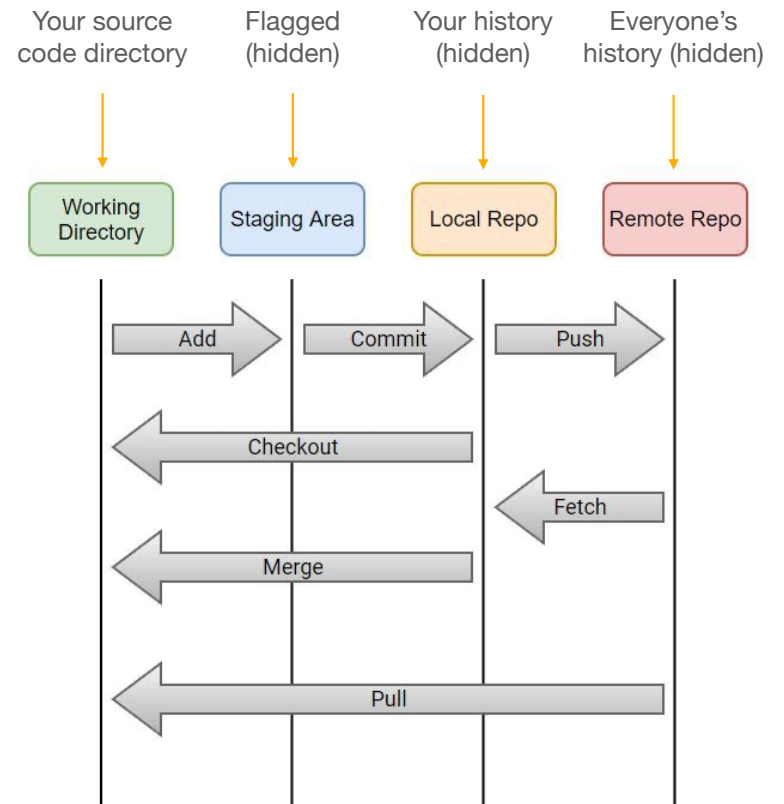
Version control systems replace the common practice of making backups of your source files. (e.g. app.c.1, app.c.2 and so on).

Git Basics

How does it work?

Git is designed around these core concepts:

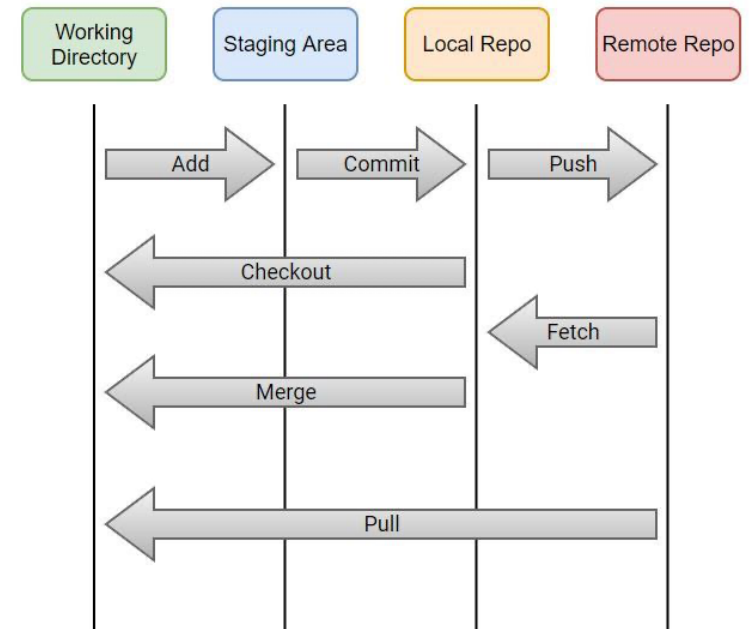
- **Working Directory:** A copy of your repository, where you will make your changes before saving them in the repository.
- **Staging Area:** A logical collection of changes from the working directory that you want to collect and work on together (e.g. it might be a feature that resulted in changes to multiple files that you want to save as a single change).
- **Repository:** The location of the canonical version of your source code (“Local Repo” in this diagram).



A repository can be local or remote:

- A **local repository** is where you might store projects that you don't need to share with anyone else (e.g. these notes are in a local git repository on my computer).
- A **remote repository** is setup on a central server, where multiple users can access it (e.g. GitLab, GitHub effectively do this, by offering free hosting for remote repositories).

You can move changes from the local to remote repository using “push” and “fetch” commands.



A local repository is only available on your local machine. You need a remote repository if you want to share your code with someone else.

Local Git Commands

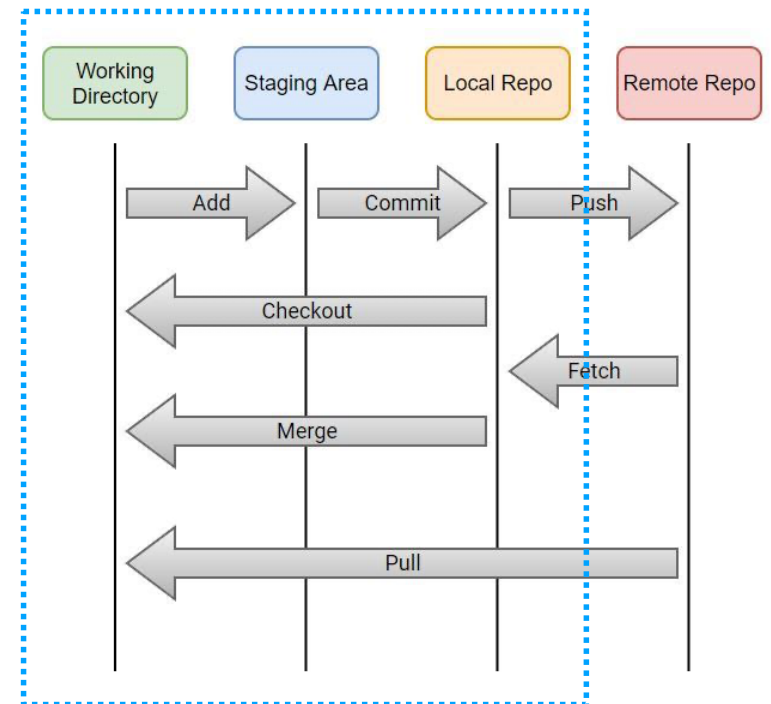
These commands use a **local** staging area and repository.

Command	Description	Example
git init	Create a new repository in the current directory.	<pre>\$ mkdir repo; cd repo \$ git init Initialized empty Git repository in /repo/.git/</pre>
git add	Add a file to the staging area	<pre>\$ vim readme.md \$ git add readme.md</pre>
git commit	Commit all staged files to the repo	<pre>\$ git commit -m "Added readme" [master (root-commit) d3c834b] Added readme 1 file changed, 0 insertions(+), 0 deletions(-) create mode 100644 readme.md</pre>
git status	Display the status of the current staging area.	<pre>\$ git status On branch master nothing to commit, working tree clean</pre>
git checkout	Checkout a specific commit to this working area. Can use to revert a file.	<pre>\$ git checkout main.kt Updated 1 path from index.</pre>

Local Workflow

A local Git workflow looks like this:

1. **Create a project directory** for your source code.
2. **Initialize a git repository** in this directory (`git init`). This doesn't change your source code, but adds a hidden `.git` directory to track it as a repository.
3. **Make changes to your source code** in your favourite editor (e.g. add a new feature, fix a bug!).
4. **Add the changed files to your staging area** (`git add`). Commit the files in the staging area to save to the repository (`git commit`). This two-step process ensures that these files are tracked and versioned as a single change.
5. **Check that your changes have been saved** by using `git status`.



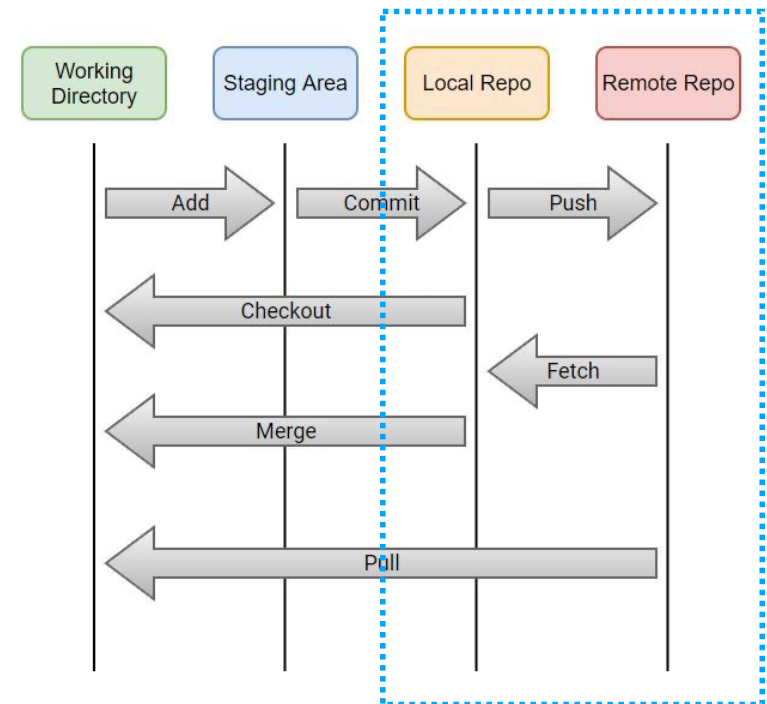
Using a Remote Repository

You can use Git locally without any restrictions. However, we often want a remote repository:

- This provides a single “source of truth” that contains everyone’s changes (and which we can backup!)
- It helps us coordinate changes with our team (i.e. if I make a change, it gives me a mechanism to share that change with everyone else).

To work with a remote repository:

- We need to setup a remote repository, or use an existing hosting site (e.g. GitLab, GitHub).
- We add a connection between local and remote repositories.
- We continue to make changes locally, but then “push” the changes to the remote as an additional step (`git push`).



Remote Git Commands

Command	Description	Example
<code>git clone</code>	Clone the remote repository to a local directory.	<pre>\$ git clone https://git.uwaterloo.ca/j2avery/cs349-public.git repo Cloning into 'repo'... remote: Enumerating objects: 531, done. remote: Counting objects: 100% (531/531), done. remote: Compressing objects: 100% (280/280), done. remote: Total 2702 (delta 209), reused 320 (delta 100), pack-reused 2171 Receiving objects: 100% (2702/2702), 7.30 MiB 13.00 MiB/s, done. Resolving deltas: 100% (939/939), done.</pre>
<code>git pull</code>	Merge changes into the local repo.	<pre>\$ cd repo \$ git pull Already up to date.</pre>
<code>git remote</code>	Modify the remote connection.	<pre>\$ git remote origin</pre>

Remote Workflow

Changes from previous workflow are marked with **remote**

The common workflow for working with a remote repository is similar to the local workflow:

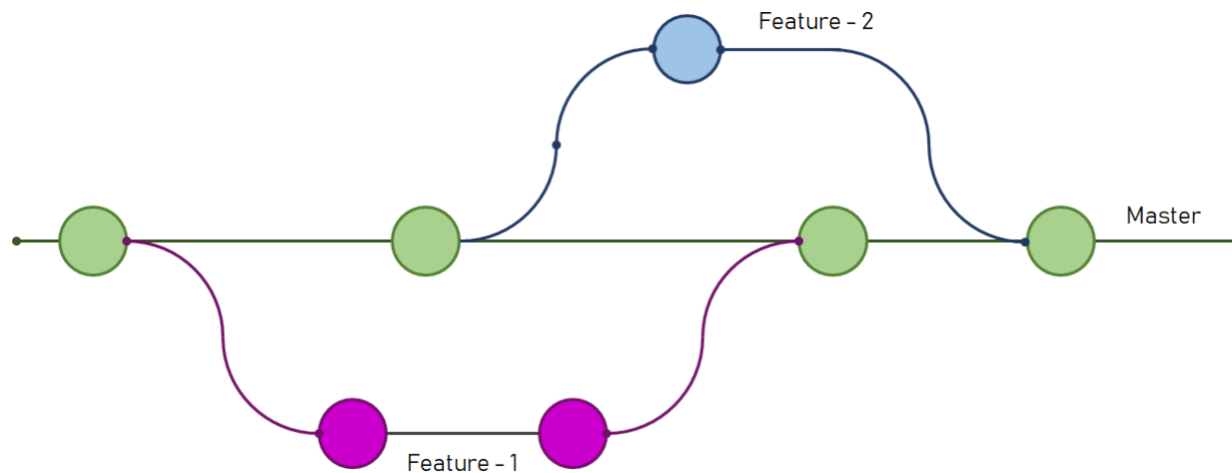
1. **Initialize a git repository** in the remote directory (e.g. in GitLab create a new project). **remote**
2. **Clone the remote repository** to create a local project directory (`git clone` using the URL of the repo that you created in the previous step). **remote**
3. **Make changes to your source code** in your favourite editor (e.g. add a new feature, fix a bug!).
4. **Add the changed files to your staging area** (`git add`). Commit the files in the staging area to save to the repository (`git commit`). This two-step process ensures that these files are tracked and versioned as a single change.
5. **Push the changes** from your local repo to the remote repo (`git push`). **remote**
6. **Check that your changes have been saved** by using `git status`.

Branching

Concept: Branching

Think of a repository as a set of commits, all in a line. The main set of commits is like a trunk of a tree. The trunk (also called Master or Main) is where commits are stored by default.

A **branch** is a fork in the tree, where we “split off” work and diverge from one of the commits. Branches diverge from a specific commit, and do not include changes that happened on the trunk after the branch occurred.



Feature branches, merged back into Main once the feature is complete and tested.

Feature Branches

We often branch to isolate our work from any other changes on the trunk. Once we have a feature implemented and tested, we can merge our changes back into the master branch.

These type of branches are called **feature branches** and isolate untested work.

A typical workflow for adding a feature would be:

- Create a feature branch for that feature.
- Make changes on your branch only. Test everything.
- (Optional) Have it code reviewed by someone on your team (see *Pull Request*).
- Switch back to master and merge from your feature branch to the master branch.

```
$ git checkout -b test // create branch
```

```
Switched to a new branch 'test'
```

```
$ vim file1.md // make some changes
```

```
$ git add file1.md
```

```
$ git commit -m "Committing changed to file1.md"
```

```
$ git checkout master // switch to master
```

```
$ git merge test // merge changes from test
```

```
Updating 09e1947..ebb5838
```

```
Fast-forward
```

```
file1.md | 136 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
1 file changed, 118 insertions(+), 18 deletions(-)
```

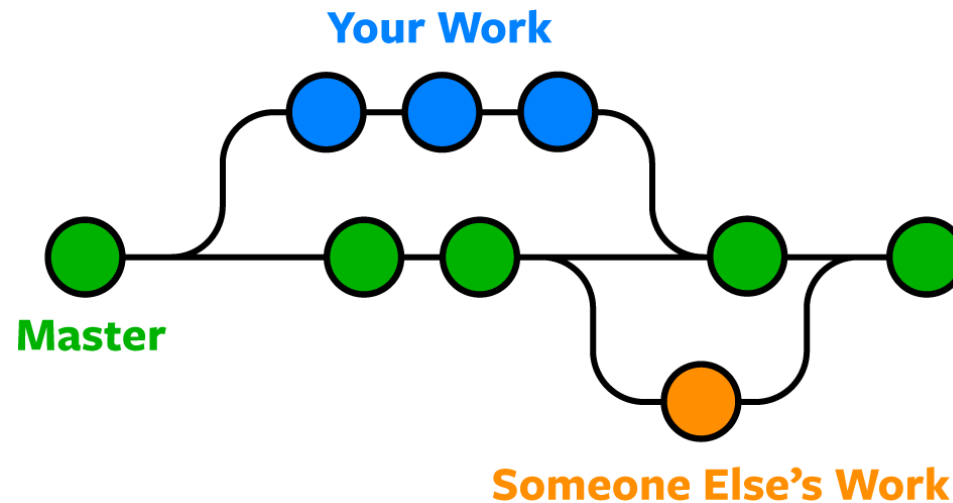
```
$ git branch -d test // remove branch (optional)
```

```
Deleted branch test (was ebb5838).
```

Collaborative Workflow

The biggest challenge when working with multiple people on the same code is that you all may want to make changes to the code at the same time. Git greatly simplifies the process.

Git uses **branches** to isolate changes from one another. You think of your source code as a tree, with one main trunk. By default, everyone in git is working from the “trunk”, typically named master or main. A branch lets you work on a separate version of the source code.

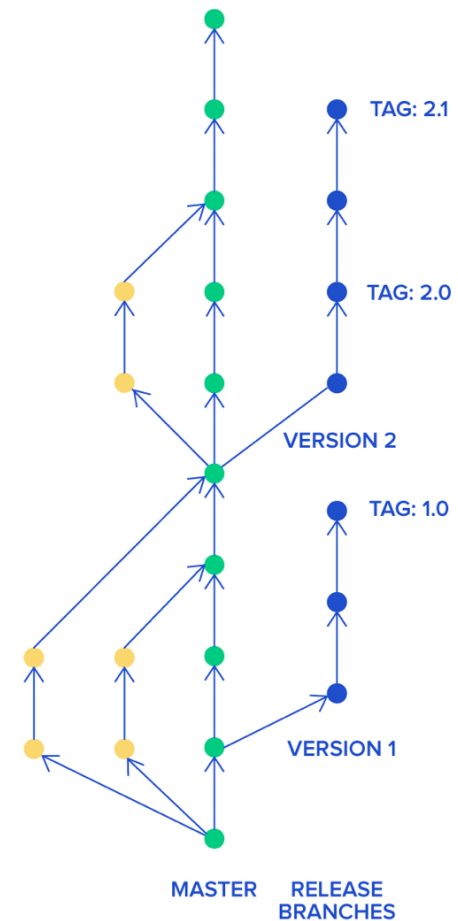


Branching Strategies

How do you coordinate branches?

There are different approaches that have been taken, but some common ideas:

1. Create feature branches for development.
2. Merge changes from feature branches to trunk; the main trunk should always build properly.
 1. Best practice: have tests on main that will automatically execute when you merge.
 2. You should always be ready to release from trunk.
3. Release from main branch.



Branching Models: Git Flow

In the Git flow model, you have two main branches:

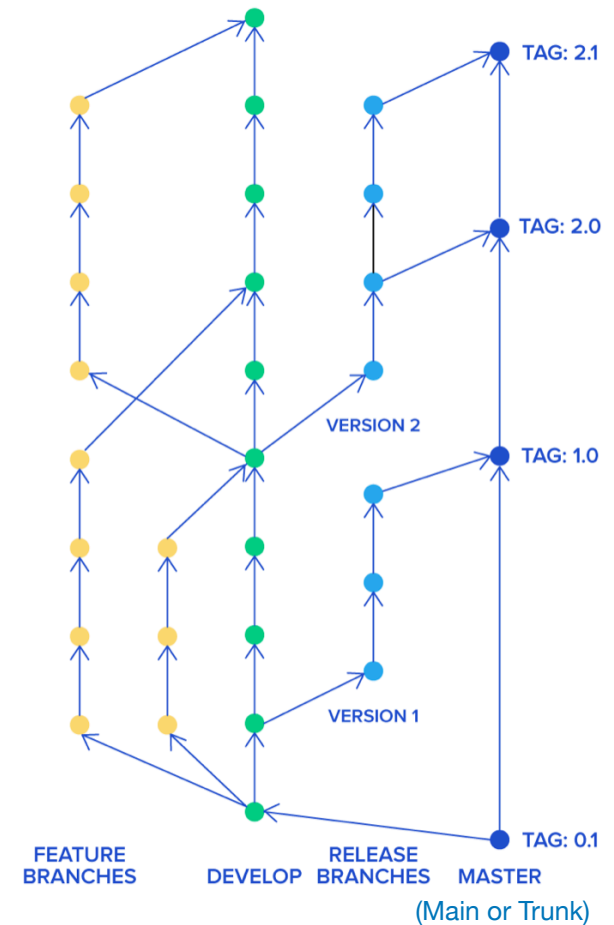
- Develop where all development takes place. Strictly controlled.
- Main or trunk that is only used for release.

Developers create feature branches from the development branch and work on them. Once features are complete, they create pull requests and other developers review their changes.

Eventually, a collection of features are approved and merged back to the trunk (main) and released as a product version.

Characteristics

- Long-lived feature branches, mean that merges are difficult.
- Allows careful control over integrating



<https://www.toptal.com/software/trunk-based-development-git-flow>

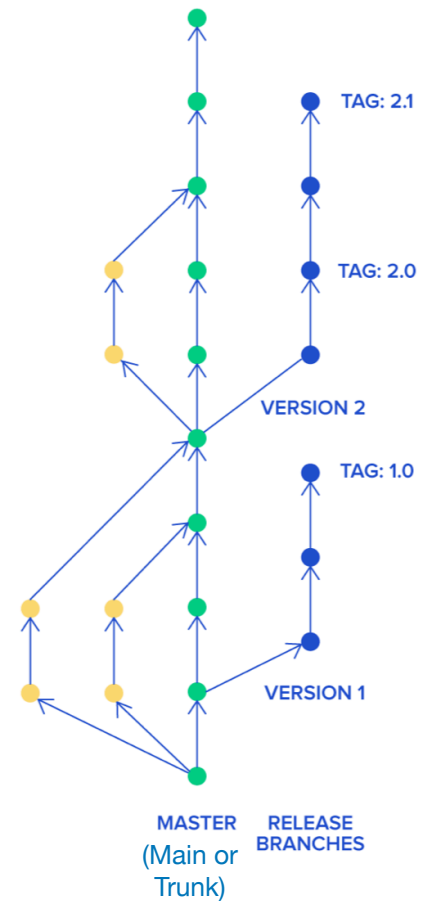
Branching Models: Trunk-Based Dev

In the trunk-based development model, all developers work on a single branch with open access to it. Often it's simply the main or trunk branch. They commit code to it and run it.

It's also common to create short-lived feature branches. Once code on their branch compiles and passes all tests, you merge straight to master.

Characteristics

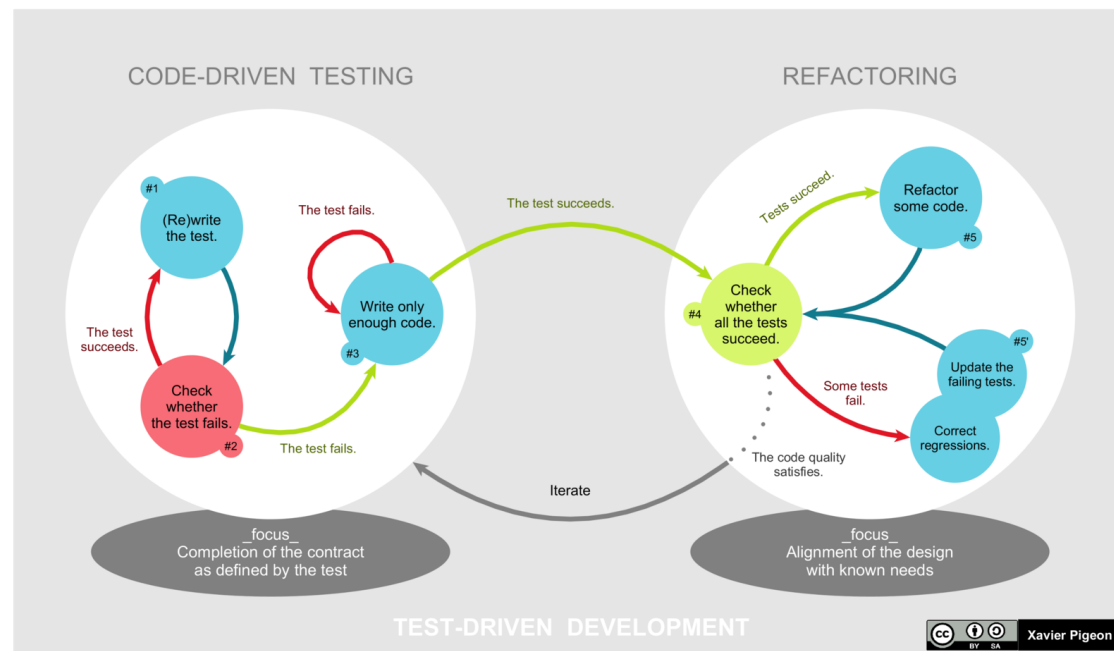
- Feature branches are short-lived.
- Development is continuous so merges are more frequent and easier to resolve.
- Integration testing becomes more important!



<https://www.toptal.com/software/trunk-based-development-git-flow>

Best Practices: TDD & Refactoring

TDD has two characters: short and correctness. TDD development cycle is very short, which make frequent commit possible. And on the other hand, the tests make sure that the code meet the requirements. The test coverage is also guaranteed with TDD. With the help of TDD, team members are able to commit to the trunk branch frequently and confidently.



Advanced Git

.gitignore

- By default, git assumes that every file and directory in your working directory is important.
 - This is not true! You will have temp files and other files you don't want to save.
- To exclude files, add the filename to a file named .gitignore in the top of your working directory.
- Files or file patterns listed in this file will be ignored by git add, status etc.

```
.DS_Store  
*.class  
build/  
out/  
www/public
```

.gitignore for the course website

Handling merges

- We've been hand-waving one of the most challenging issues in Git: merges.
- So what happens when you and one of your teammates both make changes to the same file?
 - When you commit, git merges your changes into the existing files.
 - If it can do this without any conflicts (i.e. you were working on different files, or different parts of the same files), it merges automatically. Most of the time this is what happens.
 - Occasionally you get an error when git can't resolve it.

```
! [rejected]          main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/supersites/git-er-
done.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

<https://www.raywenderlich.com/books/advanced-git/v2.0/chapters/2-merge-conflicts>

How to merge

- Use “git pull” to pull the remote changes to your working area.
- Your results will be a merge conflict. Git will tell you the source of the conflict.

```
From https://github.com/supersites/git-er-done
 7588a5f..328aa94  main    -> origin/main
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

- Edit the conflicting file and manually make the change that you want (i.e. decide which change to keep!)
- Commit the changes and “git push” to the remote repo.

Editing conflicts

- When editing the conflicting files, Git will show you both sets of proposed changes.
- You are expected to manually edit the file into a “good” state.

```
<body>
  <h1>magicSquareJS</h1>
<<<<<< HEAD
  <section>
    <input type="text" placeholder="Size" id="magic-square-size" />
    <a href="#" id="magic-square-generate-button">Generate Magic Square</a>
    <pre id="magic-square-display">
||||||| 69670e7
  <section>
    <input type="text" placeholder="Size"/>
    <a href="#">Generate Magic Square</a>
    <pre>
=====
  <section class="box">
    <input type="text" class="flex-item" placeholder="Size"/>
    <a href="#" class="flex-item btn" >Generate Magic Square</a>
    <pre class="flex-item" >
>>>>>> yUI
  </pre>
  <div id="validation" class="flex-item" ></div>
```

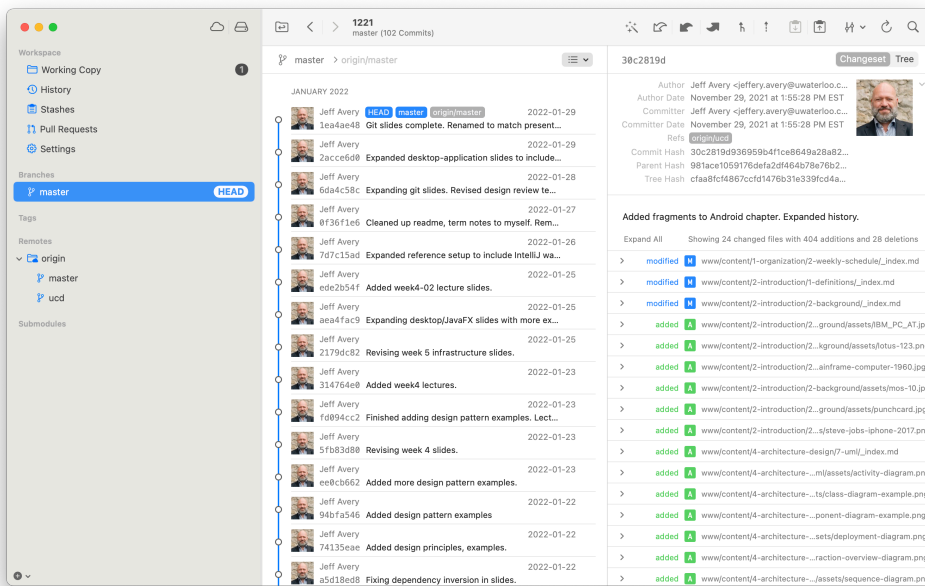
Stashes

- Sometimes you have changes that you're not ready to commit, but you want to look at a different version.
- You can stash your changes temporarily and then restore them later.

Command	Description	Example
<code>git stash</code>	Stash the current changes.	<code>\$ git stash</code>
<code>git stash list</code>	Show any stashes.	<code>\$ git stash list</code>
<code>git stash pop</code>	Restore the stash to the working directory.	<code>\$ git stash pop</code>

Git Clients

Numerous git clients exist that provide an easier alternative to the command-line.



<https://www.git-tower.com>



<https://github.com/jesseduffield/lazygit>



<https://xkcd.com/1597/>