Version Control with Git

Computer Science Practice and Experience: Development Basics
CS1XC3

Professor: Kevin Browne
E-mail: brownek@mcmaster.ca

Code changes a lot over the lifespan of real-world projects.

Source: https://lh3.googleusercontent.com/proxy/TGEdeuZBjAxv-5hU4LG9OMcR0kWLSx7C5u_lyv8II5Q2Cm39VyHKr7YdK0FVtc-bq7h5ExjEpvl_xyYrvgdAaD93-6P0qou-QZZLRy3iHj58irYKuhUIxqkSg7Syefp_FwWVv3HOV4imsX2Qnz-fOUEZKidTwsXaIntEYA

Managing your own code changes is bad.

Now imagine hundreds of developers working on a project at once…

Source: https://www.aberdeen.com/wp-content/uploads/2016/09/marketing_and_sales_gridlock.jpg

Coding as a team

- One approach might be to have developers work on separate areas of code

- In practice, this is very often exactly what happens
  - One developer works on one component (e.g. a library in C), another developer works on another component, etc.
  - Advantage: developers don't make changes to overlapping areas of code
  - Advantage: developers build expertise about an area of code, can make changes and updates faster

- Some teams will have developers that "own" sections of code

Coding as a team

- Problem: what about areas of code where components interact and connect?
  - At a certain point components need to work together to carry out work
- Problem: what if multiple developers need to work on the same component?
  - Some components are larger, more complex
  - Some components are very important, may need to change rapidly

Source: https://upload.wikimedia.org/wikipedia/commons/5/5d/York_Region_Transit_1502.JPG

Bus factor

- Is it ***really*** a good idea to have a developer "own" a section of code? What if they get hit by a bus?
  - Or perhaps less dramatically... what if they leave the project?

- "The **bus factor** is a measurement of the risk resulting from information and capabilities not being shared among team members, derived from the phrase 'in case they get hit by a bus'." - Wikipedia

- Project managers will aim to "lower the bus factor" by having multiple developers able to modify different sections of code

Coding as a team

- So while as a practical matter on most projects...
  - Developers will "own" sections of code
  - Developers will work independently on sections of code
- ...it's also sometimes impossible, and very often inadvisable to rely on this approach alone for coordinating code changes
- We're going to need an additional tool to help us manage changes to code

Version control systems

- **Version control systems** manage changes to source code and related files in **repositories**

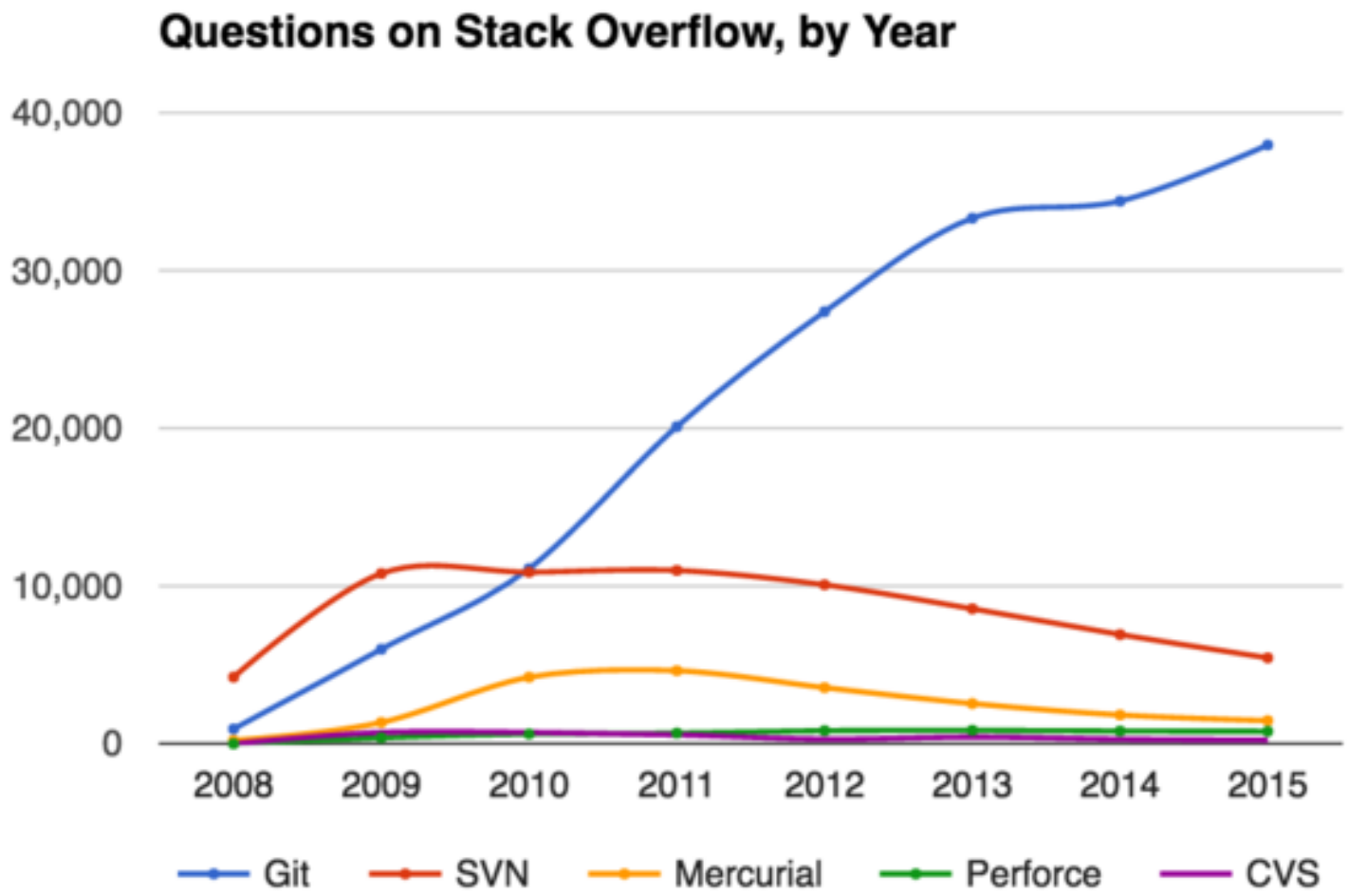
- Version control systems keep track of different numbered versions of source code files
  - Developers have the ability to save changes to files and create new versions in the process
  - Developers can also access older and alternative versions of files
  - So no more saving your files as as1_old.c, as1new.c, as1new2.c, etc. :-)

- Developers are able to access shared repositories of files, rather than sending zip folders back and forth

Version control systems

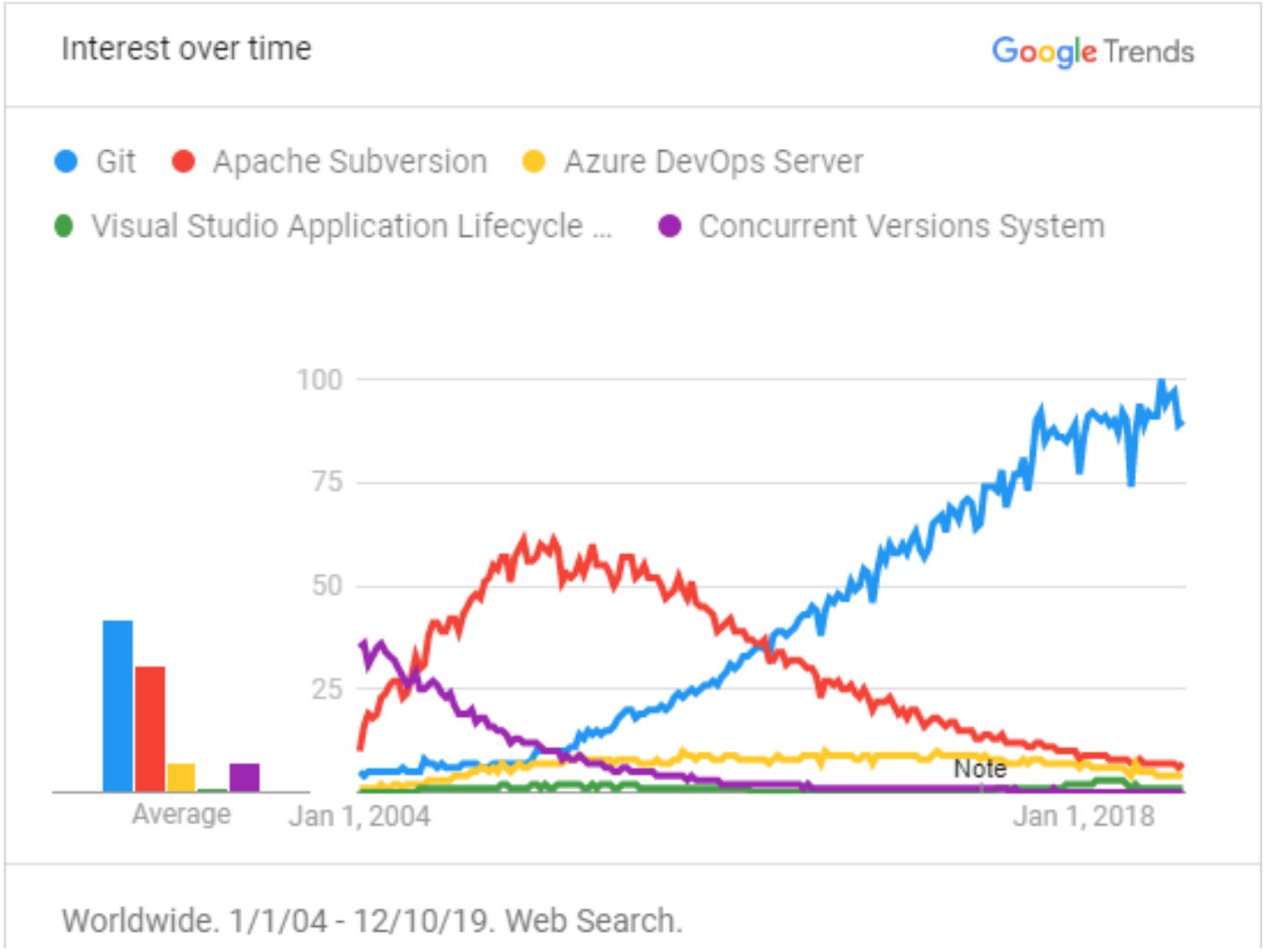
- Earlier version control systems included Revision Control System (RCS) and Concurrent Versions System (CVS)
  - CVS was essentially built on top of RCS
  - The instructor learned with CVS... and now feels old! :-(
- These were largely superceded by Subversion (SVN) which added additional features and improvements
  - Very popular from about 2001-2012
  - Still used in some projects
- Git is now the industry standard version control system

Version control usage statistics



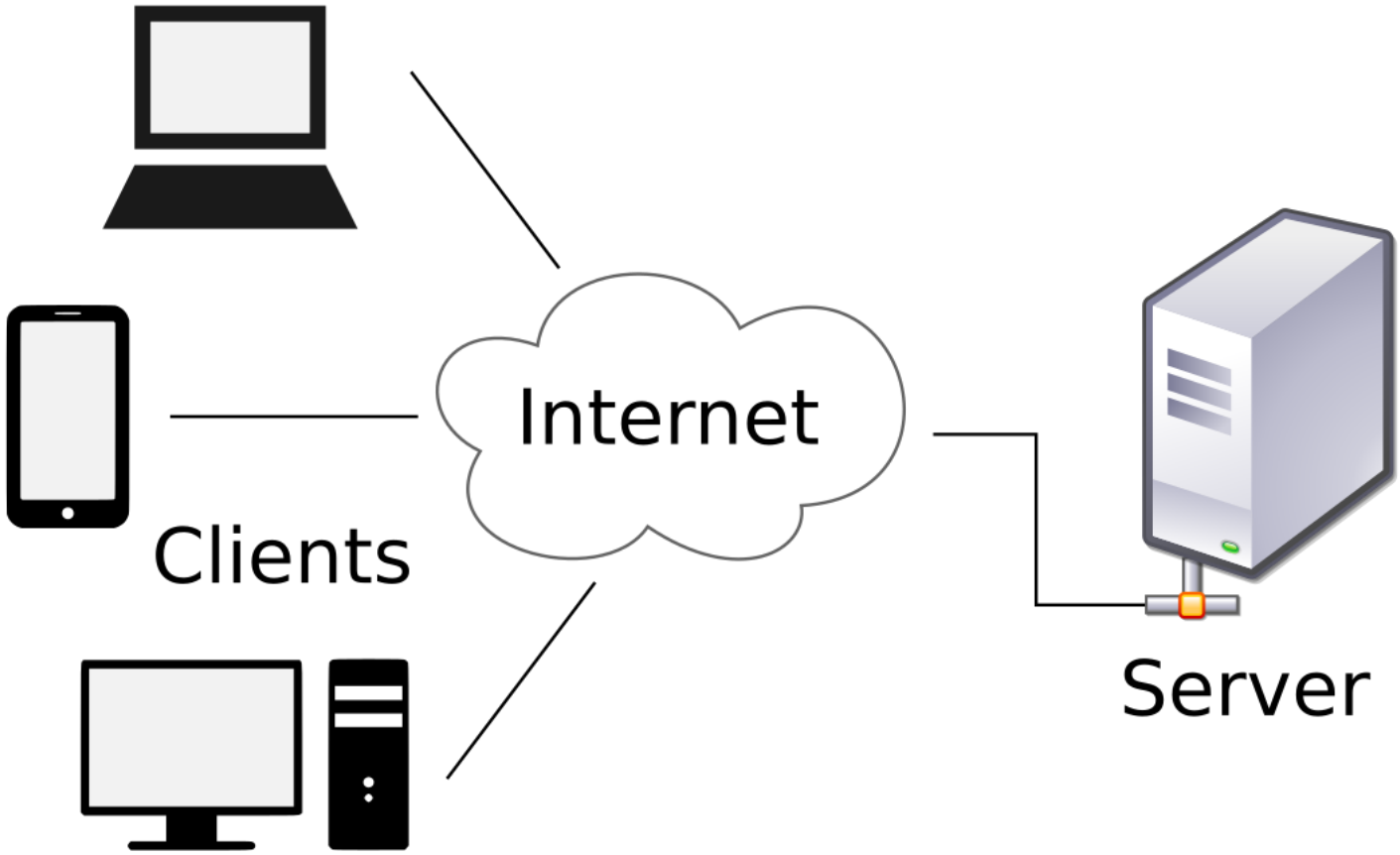
Source: https://rhodecode.com/static/img/reports/stackoverflow-trend_mini.png

Version control usage statistics



Source: https://miro.medium.com/max/1092/1*w7MWPimDioDt300sXspcZg.png

Server-client model



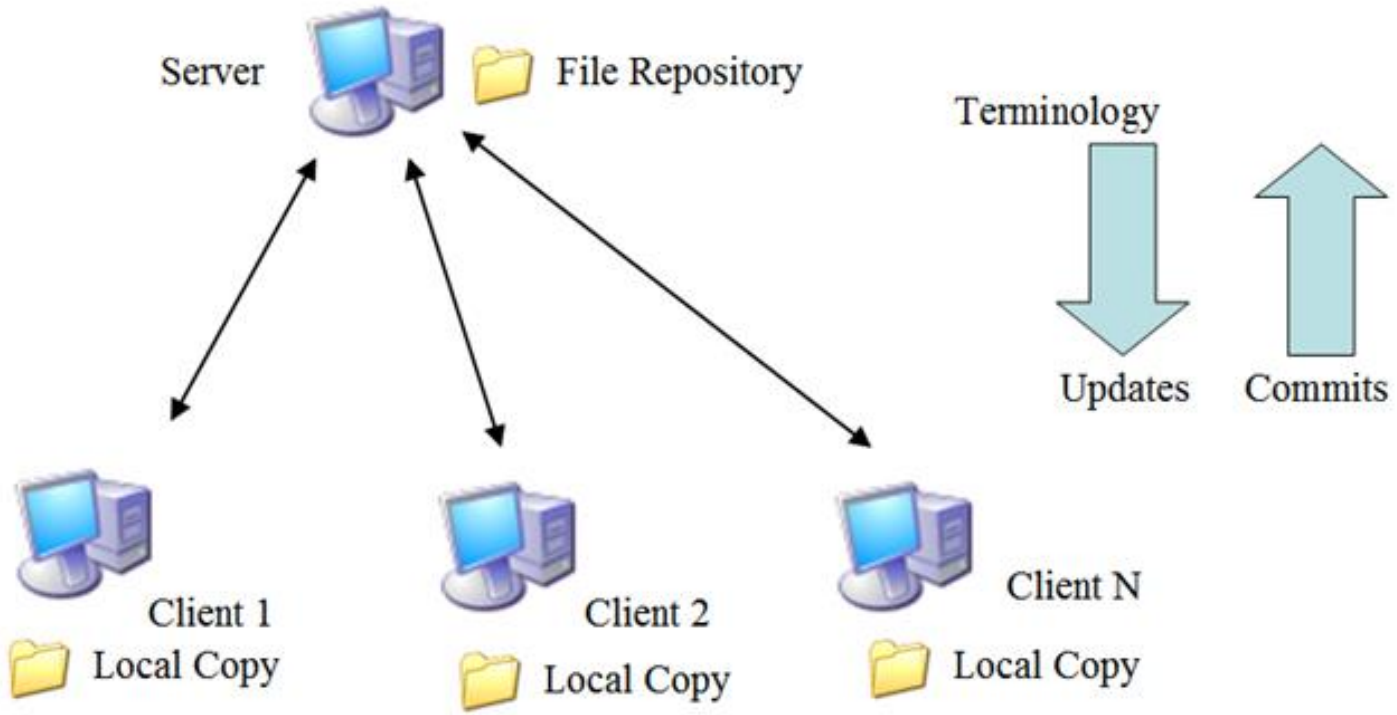
Source: https://upload.wikimedia.org/wikipedia/commons/thumb/c/c9/Client-server-model.svg/1200px-Client-server-model.svg.png

Git

- Previous version control systems followed a client-server model
  - A repository of code is kept and managed on a server
  - Developers on client machines check out code from the server (load code on to their own computer) and check in code back to the server (save code back to the server)

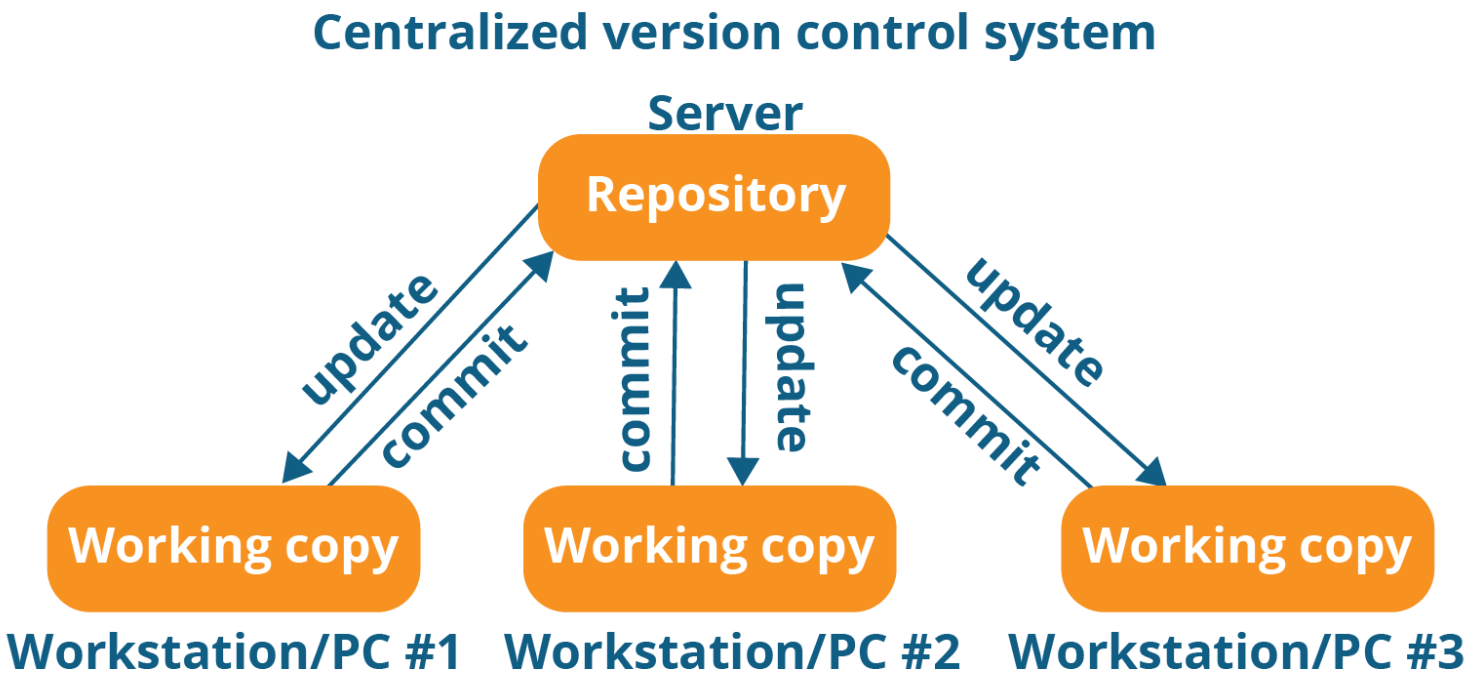
- Git is a **distributed version control** system where each developer has a fully functioning repository on their own machine

Client-server version control



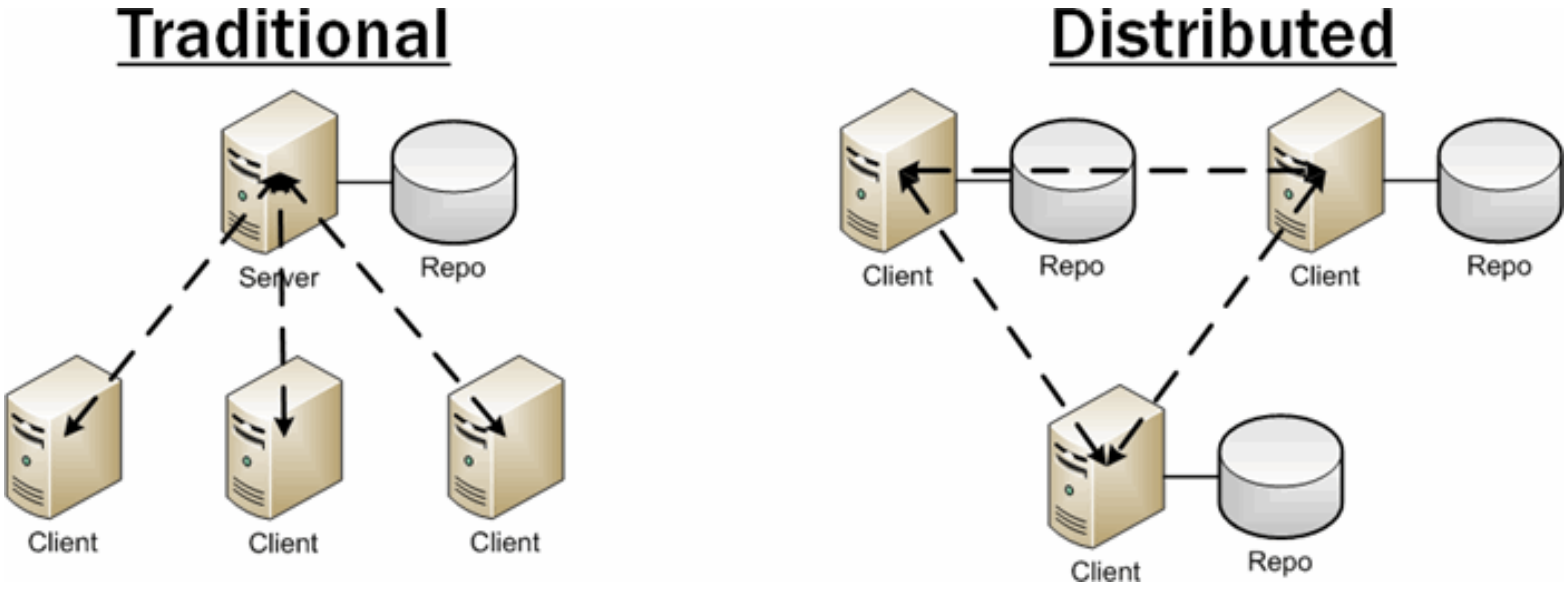
Source: https://www.bowdoin.edu/~sbarker/unix/svn.png

Client-server version control



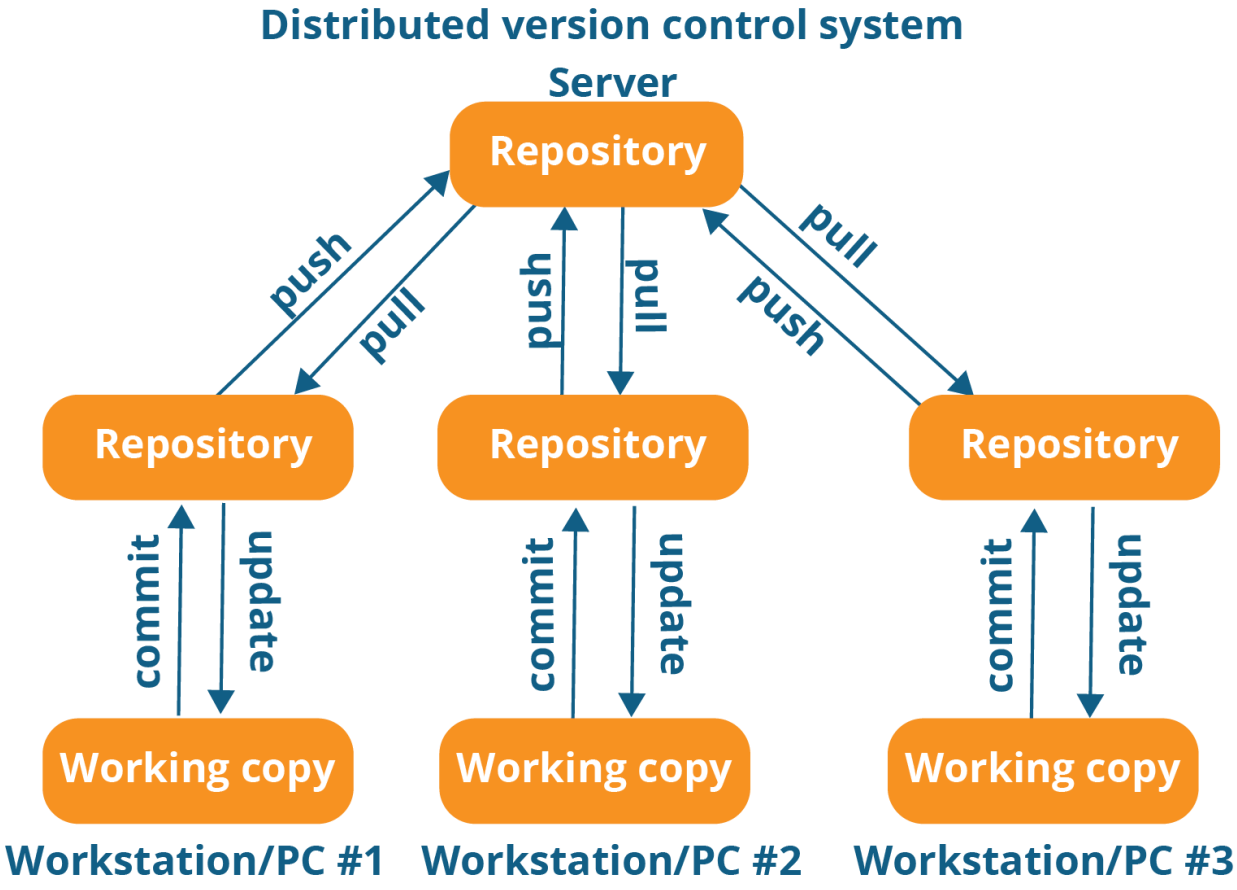
Source: https://miro.medium.com/max/1698/1*GgaGcwh5L246YcU5NVDA5A.png

In a distributed version control system, every client has a fully functioning repository of code...



Source: https://thedailywtf.com/images/201109/scm/dcm1.png

As a practical matter, even though all clients have their own repository, a server will still be used to provide a central point of coordination



Source: https://miro.medium.com/max/3396/1*CEyiDu_mQ5u9NI0Fr2pSdA.png

Git

- Git was created by Linus Torvalds in 2005
  - Linus was also the creator of the Linux kernel
- Linus had a reputation for being a jerk to other developers (seriously)
  - So he named it "git" which is an "unpleasent person" in British slang
  - Claims now both git ***and*** Linux are named after him!

Source: https://en.wikipedia.org/wiki/Linus_Torvalds

Getting started with Git

- Git can be downloaded from
  - https://git-scm.com/downloads
- There are GUI interfaces, we'll be using the command-line version
- It's fine to use a GUI interface in general, though you'll find some programmers are snobby about this
- If you're going to be a professional software developer, it's expected you understand the command-line version

Getting started with Git

- Git **repositories** store changes to files
- We create a repository by **initializing** a repository
- We initialize a repository with the command **git init**

```
[brownek@pascal code]$ git init
Initialized empty Git repository in /home/brownek/code/.git/
[brownek@pascal code]$
```

Where is the repository?

You can use **ls –a** to see the hidden .git folder that is created.

We never touch the contents of this folder, but this is the repository.

```
[brownek@pascal code]$ ls
[brownek@pascal code]$ ls -a
.  ..  .git
[brownek@pascal code]$ cd .git
[brownek@pascal .git]$ ls -a
.  ..  branches  config  description  HEAD  hooks  info  objects  refs
[brownek@pascal .git]$
```

Using a repository

- To use the repository, we need to **add** files
  - When we add files, we are telling git that we want the repository to keep track of these files
  - When we make changes to files, we add them again to add them to the **staging area**

- To save a version of our files to the repository, we **commit** the files
  - When we commit, any files in the staging area will be committed to the repository
  - A **snapshot** of the files in the staging area is taken when we commit

- Versions of files are kept track of along different **branches**, the default branch is the "master" branch

git status

- The **git status** command will show us important information such as...
  - What files are not being managed by the repository
  - What files have been changed since the last commit
  - What branch we are currently on

```
[brownek@pascal code]$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
[brownek@pascal code]$
```

Adding and committing files

- **git add filename**
  - This will add a file with filename to the repository
  - We could list multiple filenames like **git add f1 f2 f3**
  - We can add all files with **git add .**

- **git commit –m "commit log message"**
  - This will commit any files in the staging area to the repository
  - Commits are given log messages, we can use the –m argument to provide short log messages
  - Log messages are important, they should describe what changes are being committed to the repository

Adding a file

```
[brownek@pascal code]$ touch file.txt
[brownek@pascal code]$ ls
file.txt
[brownek@pascal code]$ git add file.txt
[brownek@pascal code]$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file.txt

[brownek@pascal code]$
```

Comitting a file

```
[brownek@pascal code]$ git commit -m "Adding file.txt to the repository"
[master (root-commit) cf96f78] Adding file.txt to the repository
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file.txt
[brownek@pascal code]$ git status
On branch master
nothing to commit, working tree clean
[brownek@pascal code]$
```

Notice how **git status** previously told us we had a file ready to be committed, and now after comitting is telling us we have nothing to commit.

git status will also tell us about files that haven't been added yet and are thus untracke by git, like if we create this new file another.txt...

```
[brownek@pascal code]$ touch another.txt
[brownek@pascal code]$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        another.txt

nothing added to commit but untracked files present (use "git add" to track)
[brownek@pascal code]$ ls
another.txt  file.txt
[brownek@pascal code]$
```

Log

- The repository will keep track of all commits and we can view a log of all commits with **git log**

- Commits will have a unique ID, a 40-digit hash, as well as author, date and commit log messages

```
[brownek@pascal code]$ git log
commit cf96f78572e09379d578e9d9436040dab71f46cd (HEAD -> master)
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 07:57:52 2021 -0400

    Adding file.txt to the repository
[brownek@pascal code]$
```

We can add and commit the another.txt file now...

```
[brownek@pascal code]$ git add .
[brownek@pascal code]$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   another.txt

[brownek@pascal code]$ git commit -m "Added another.txt to the repository"
[master 16bc743] Added another.txt to the repository
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 another.txt
```

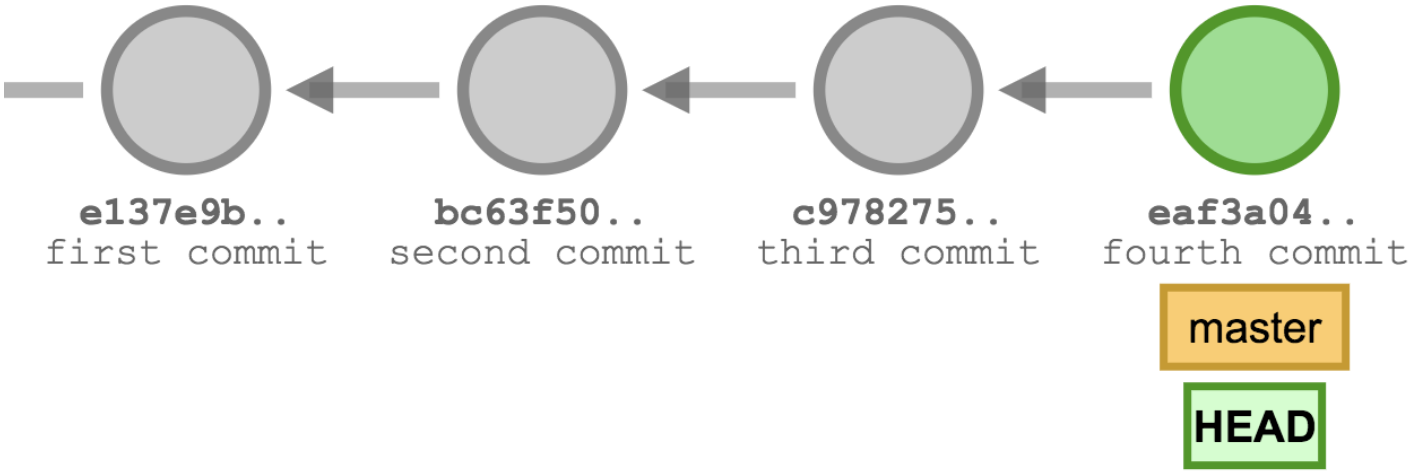
And now when we run **git log** we'll see a new commit message

```
[brownek@pascal code]$ git log
commit 16bc7433d7ffbf88ec1df39656ad28534295f90f (HEAD -> master)
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:15:37 2021 -0400

    Added another.txt to the repository

commit cf96f78572e09379d578e9d9436040dab71f46cd
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 07:57:52 2021 -0400

    Adding file.txt to the repository
[brownek@pascal code]$
```

What is head and master referring to?

Git commits

- As we create each commit, we essentially create a new node in a chain of commits
  - The head is the most recent commit of a branch

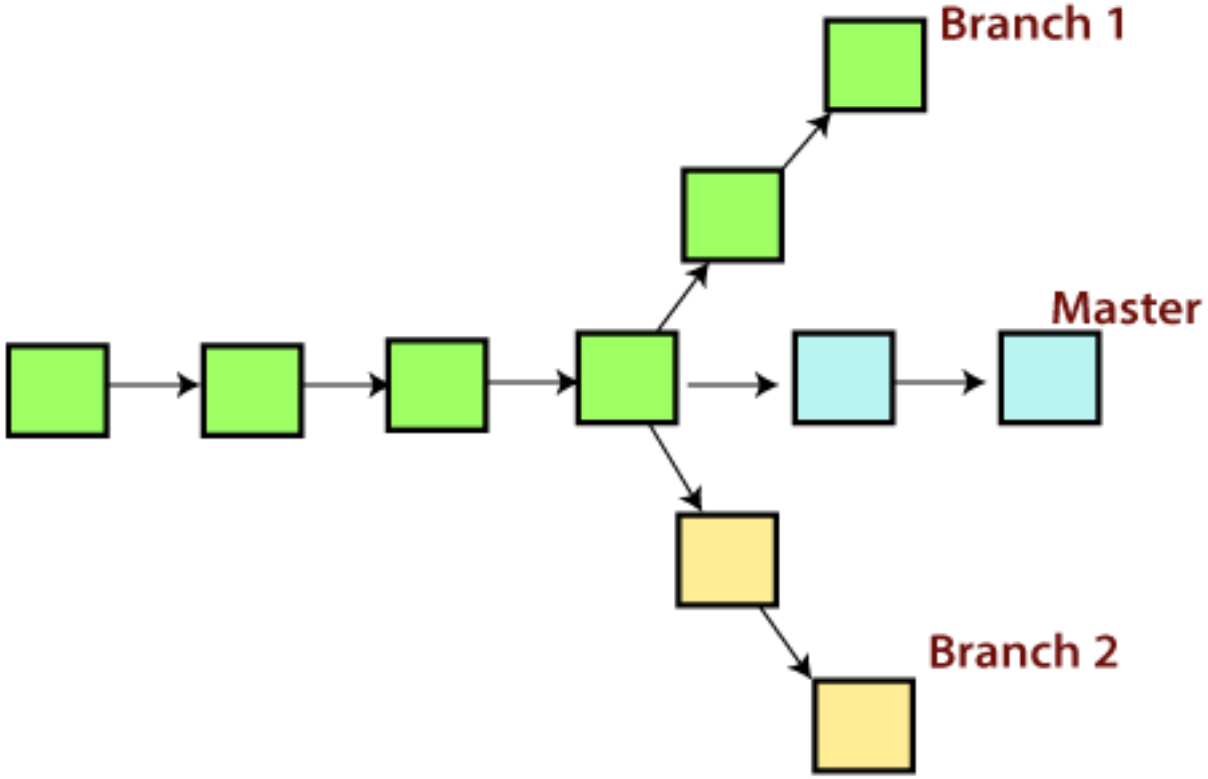


Source: https://miro.medium.com/max/2540/1*tFGUQ9I0DYY3NBk2ZBY97A.png

Git branches

- Git repositories are split up into **branches**
  - The default branch all repositories have when they are created is the "master" branch
  - The master branch is the mainline branch, in practice it is expected to be a working copy of the program
  - We can create other branches and perform new commits that will not effect the master branch
  - We can **merge** branches together to merge changes
- Typical workflow is to create a branch for a new feature or development in general
  - And then after the feature is done and working, merge the change to the master branch

Branches



Source: https://static.javatpoint.com/tutorial/git/images/git-branch.png

Git branches

- **git branch branch_name**
  - Creates a branch with branch_name
  - Does not automatically switch to this branch

- **git switch branch_name**
  - Switches to the branch with branch_name
  - New commits will now be added to this branch

Creating a "dev" branch and switching to it…

```
[brownek@pascal code]$ git branch dev
[brownek@pascal code]$ git status
On branch master
nothing to commit, working tree clean
[brownek@pascal code]$ git switch dev
Switched to branch 'dev'
[brownek@pascal code]$ git status
On branch dev
nothing to commit, working tree clean
[brownek@pascal code]$
```

Let's say we modify file.txt...

```
Add some content to file.txt.
~
~
~
~
~
~
~
```

And then add and commit our change…

```
[brownek@pascal codenew]$ vi file.txt
[brownek@pascal codenew]$ git add .
[brownek@pascal codenew]$ git status
On branch dev
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file.txt

[brownek@pascal codenew]$ git commit -m "Modified file.txt on the dev branch"
[dev d4fccbf] Modified file.txt on the dev branch
 1 file changed, 1 insertion(+)
```

Now when we run git log, we'll see the new head is for the dev branch…

```
[brownek@pascal codenew]$ git log
commit d4fccbf75d4f0c9725263389a16da38d001867d4 (HEAD -> dev)
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:53:45 2021 -0400

    Modified file.txt on the dev branch

commit a689dc297b8fa56179c86512624cdafae30465b5 (master)
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:52:56 2021 -0400

    Added another.txt to the repository

commit 806f144a1fb19c246b18c669ee574c0348de6c6b
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:51:58 2021 -0400

    Adding file.txt to the repository
```

If we switch to master and run git log we won't see the newest commit…

```
[brownek@pascal codenew]$ git switch master
Switched to branch 'master'
[brownek@pascal codenew]$ git log
commit a689dc297b8fa56179c86512624cdafae30465b5 (HEAD -> master
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:52:56 2021 -0400

    Added another.txt to the repository

commit 806f144a1fb19c246b18c669ee574c0348de6c6b
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:51:58 2021 -0400

    Adding file.txt to the repository
[brownek@pascal codenew]$
```

git log

- git log by default will show all previous commits relevant to the current branch
  - In the case of our dev branch, it's showing us the previous commits on the master branch because those commits are the ancestors of the current branch head

- We can view all commits across all branches with **git log --all**

Running **git log --all** while on the master branch, we can see the dev branch commits now too

```
[brownek@pascal codenew]$ git log --all
commit d4fccbf75d4f0c9725263389a16da38d001867d4 (dev)
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:53:45 2021 -0400

    Modified file.txt on the dev branch

commit a689dc297b8fa56179c86512624cdafae30465b5 (HEAD -> master)
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:52:56 2021 -0400

    Added another.txt to the repository

commit 806f144a1fb19c246b18c669ee574c0348de6c6b
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:51:58 2021 -0400

    Adding file.txt to the repository
[brownek@pascal codenew]$
```

Let's modify and commit another.txt on the master branch...

```
Making modifications to another.txt.
~
~
~
~
~
~
```

```
[brownek@pascal codenew]$ vi another.txt
[brownek@pascal codenew]$ git add .
[brownek@pascal codenew]$ git commit -m "modified another.txt on master branch"
[master c881fd0] modified another.txt on master branch
 1 file changed, 1 insertion(+)
```

Branches

- At this point, the master branch is **different** than the dev branch

- Before the dev branch really just had more commits than the master branch, but it effectively contained the master branch (in terms of all previous commits)

- With new commits added to the master and dev branch, we now really have two meainginfully different branches

Visualizing branches

- We can visualize branches using git log
- **git log --all --graph**
  - Produces a graph like representation of the different branches in the commit log
- Let's try using this command...

```
[brownek@pascal codenew]$ git log --all --graph
* commit c881fd071c66c2e21f138c9ca06d29ea070129e9 (HEAD -> master)
| Author: brownek <brownek@pascal.cas.mcmaster.ca>
| Date:   Thu Mar 25 08:58:00 2021 -0400
|
|     modified another.txt on master branch
|
| * commit d4fccbf75d4f0c9725263389a16da38d001867d4 (dev)
|/  Author: brownek <brownek@pascal.cas.mcmaster.ca>
|   Date:   Thu Mar 25 08:53:45 2021 -0400
|
|       Modified file.txt on the dev branch
|
* commit a689dc297b8fa56179c86512624cdafae30465b5
| Author: brownek <brownek@pascal.cas.mcmaster.ca>
| Date:   Thu Mar 25 08:52:56 2021 -0400
|
|     Added another.txt to the repository
|
* commit 806f144a1fb19c246b18c669ee574c0348de6c6b
  Author: brownek <brownek@pascal.cas.mcmaster.ca>
  Date:   Thu Mar 25 08:51:58 2021 -0400

      Adding file.txt to the repository
```

Merging changes

- What if we want to merge changes from the dev branch into the master branch
- While on the master branch we can use the command: **git merge dev**
- Changes from the dev branch will be merged into the master branch
- Another useful command: **git branch**
  - Will show all branches, highlights current branch with *

Merging the dev branch…

```
[brownek@pascal codenew]$ git branch
  dev
* master
[brownek@pascal codenew]$ git merge dev
Merge made by the 'recursive' strategy.
 file.txt | 1 +
 1 file changed, 1 insertion(+)
[brownek@pascal codenew]$ git branch
  dev
* master
```

Merging branches

- When branches are merged a commit message is required
  - By default git will generally launch **vi** to create commit messages
- Remember when using vi you can escape "writing mode" by hitting escape
  - And then :wq will allow you to save and quit

Now changes from both branches are present…

```
[brownek@pascal codenew]$ cat file.txt
Added some content to file.txt.
[brownek@pascal codenew]$ cat another.txt
Making modifications to another.txt.
```

Merging the branches creates a new commit…

```
[brownek@pascal codenew]$ git log --all
commit c6fb6d1f3cac519cd2d7faf222de470a32c6903b (HEAD -> master)
Merge: c881fd0 d4fccbf
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 09:05:06 2021 -0400

    Merge branch 'dev'

commit c881fd071c66c2e21f138c9ca06d29ea070129e9
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:58:00 2021 -0400

    modified another.txt on master branch

commit d4fccbf75d4f0c9725263389a16da38d001867d4 (dev)
Author: brownek <brownek@pascal.cas.mcmaster.ca>
Date:   Thu Mar 25 08:53:45 2021 -0400

    Modified file.txt on the dev branch
```

How did the merge work?

- In this case, a different file was modified in each branch, and so a merge was trivial... new versions of the files from the old branch can overwrite the old versions from master

- What if the ***same*** files are modified?
  - git will attempt to merge the file content
  - This may or may not "work"... if some lines were added to different sections of each file it may be OK
  - If the same lines were modified or lines in the same place of the file are modified we may have a **conflict**
  - A conflict is when git cannot figure out how to merge files together
  - Conflicts are up to the developer to figure out!

- We'll talk more about conflicts tomorrow