

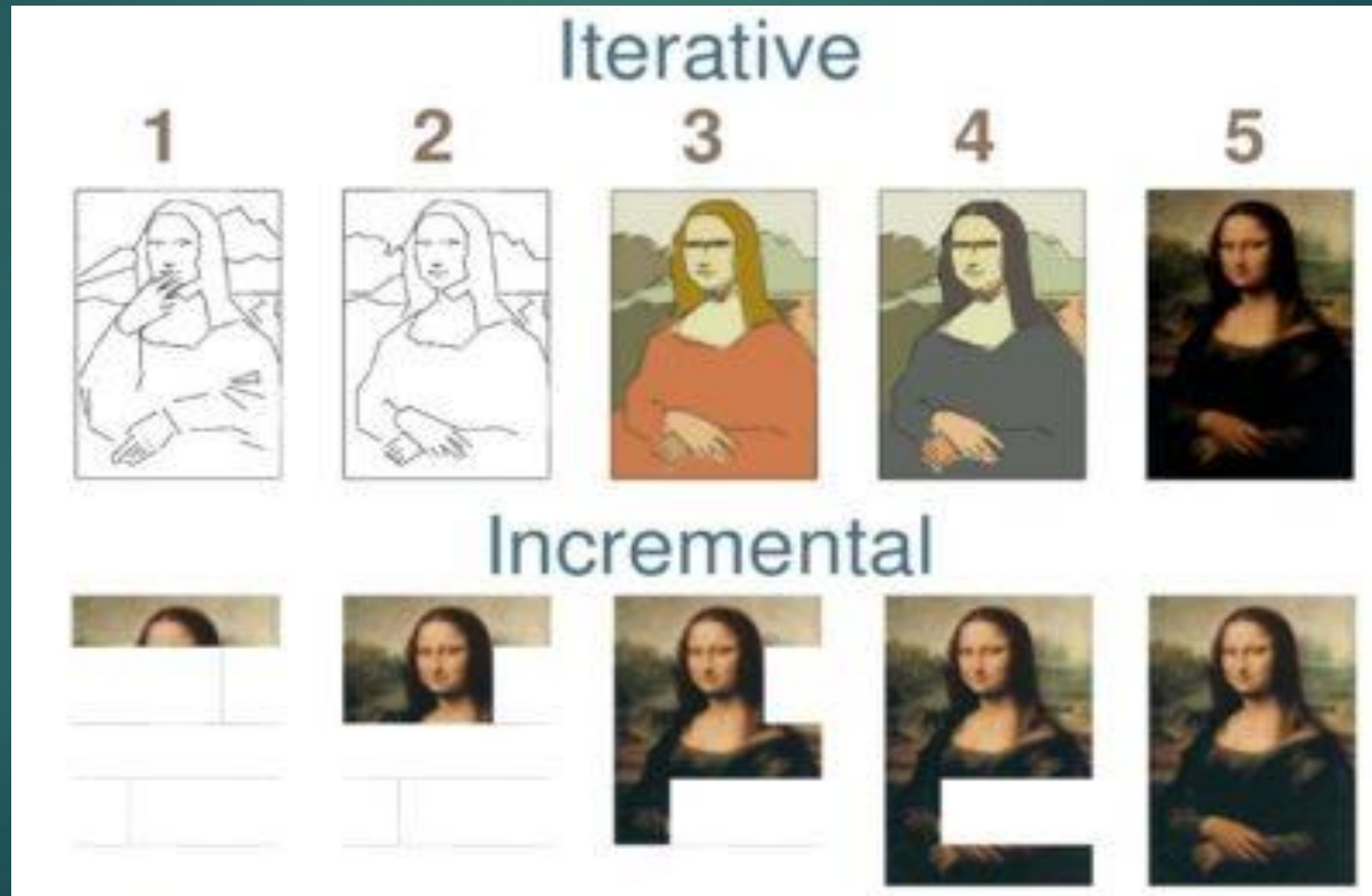
# Source Code Management + Git

CS 3140 – LECTURE 03

# Gradual Development

- ▶ Homework 1 has multiple components
  - ▶ Handling command line arguments
  - ▶ Reading in CSV data
  - ▶ Using that data to apportion representatives
  - ▶ Displaying that output
- ▶ You aren't going to sit down and write all this code in one sitting
- ▶ So how can we approach this?

# Incremental vs iterative



# Version Control



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

# All Software Has Multiple Versions

- ▶ Different releases
  - ▶ More features
  - ▶ Bug fixes
- ▶ New platforms
- ▶ Test releases
  - ▶ Alpha, beta, final

# All Software Has Multiple Versions

- ▶ Every time you edit a file, you create a new version
- ▶ When you are working alone, this is fine
- ▶ But what if working with a team of 4 people?
  - ▶ How do you resolve conflicts?
- ▶ Conflicts certain in open source projects
  - ▶ Hundreds of committers

# Version control

- ▶ Version control tracks multiple file versions
- ▶ Version control allows
  - ▶ Multiple versions of a file can exist simultaneously
  - ▶ Older versions of files can be recovered
- ▶ Mainstay in software development
  - ▶ `git` is the trendiest right now – We will use this one
  - ▶ `svn` is worth knowing about, but has \*drastically\* decreased in popularity

# Git vs. SVN

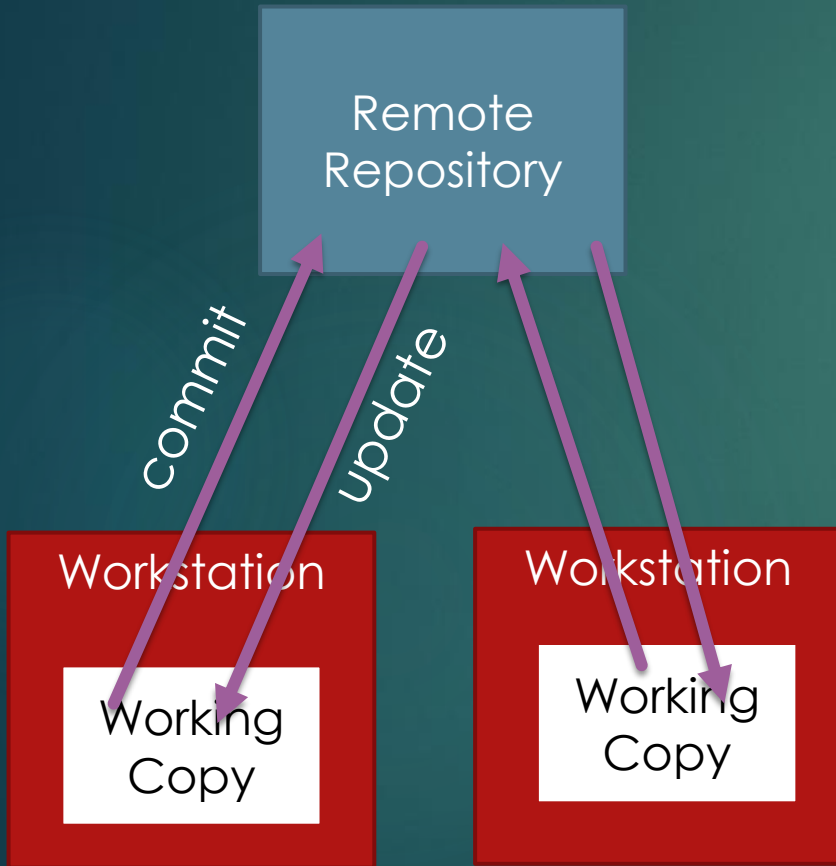
Git	vs	SVN
Used by ~90% of professional developers		Used by ~10% of professional developers
Distributed Version Control		Centralized Version Control
Can work locally, offline		Must be connected to commit
Each user has a copy of the full repository		Each user only has a copy of the trunk
Easy to fork, branch, and merge		Branching and merging is time-consuming

Source – Axosoft Blog – Ryan Pinkus - <https://blog.axosoft.com/migrating-git-svn/>



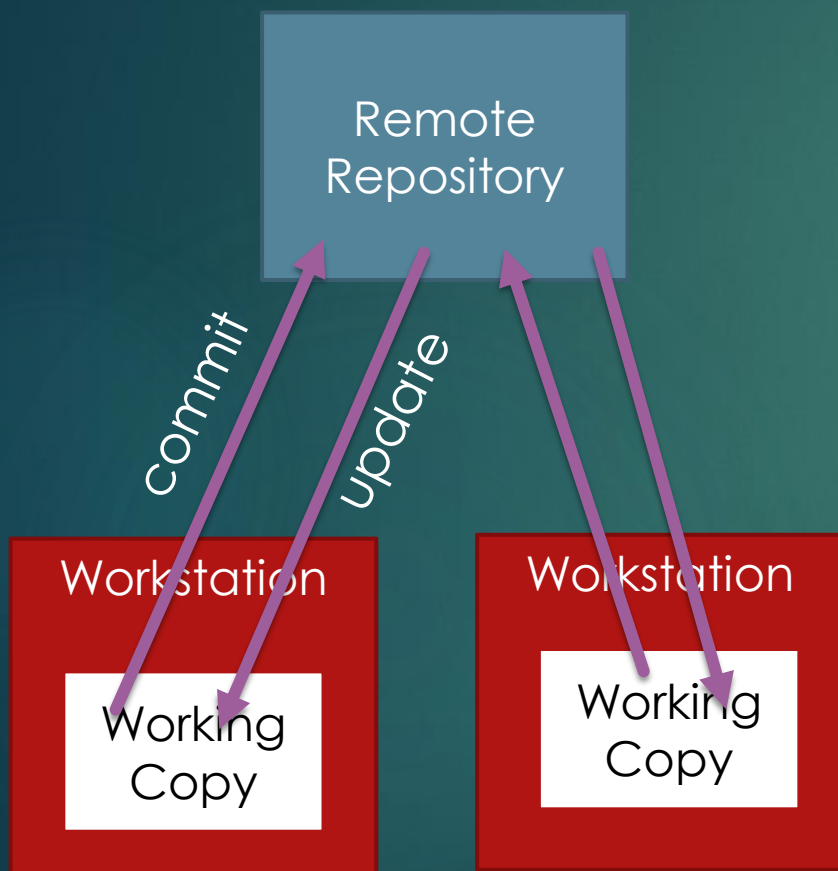
# Git vs. SVN

Centralized (SVN)

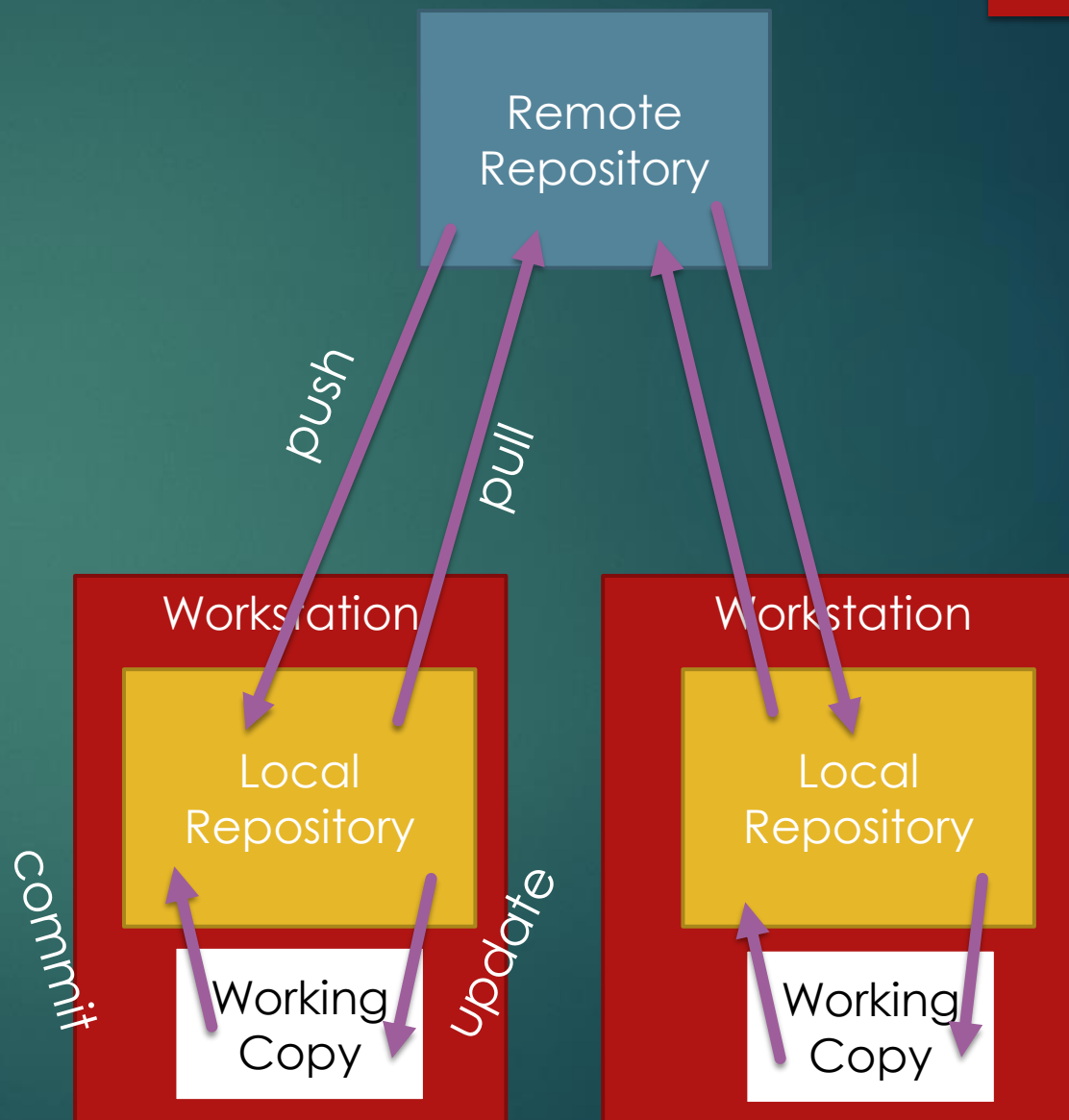


# Git vs. SVN

## Centralized (SVN)

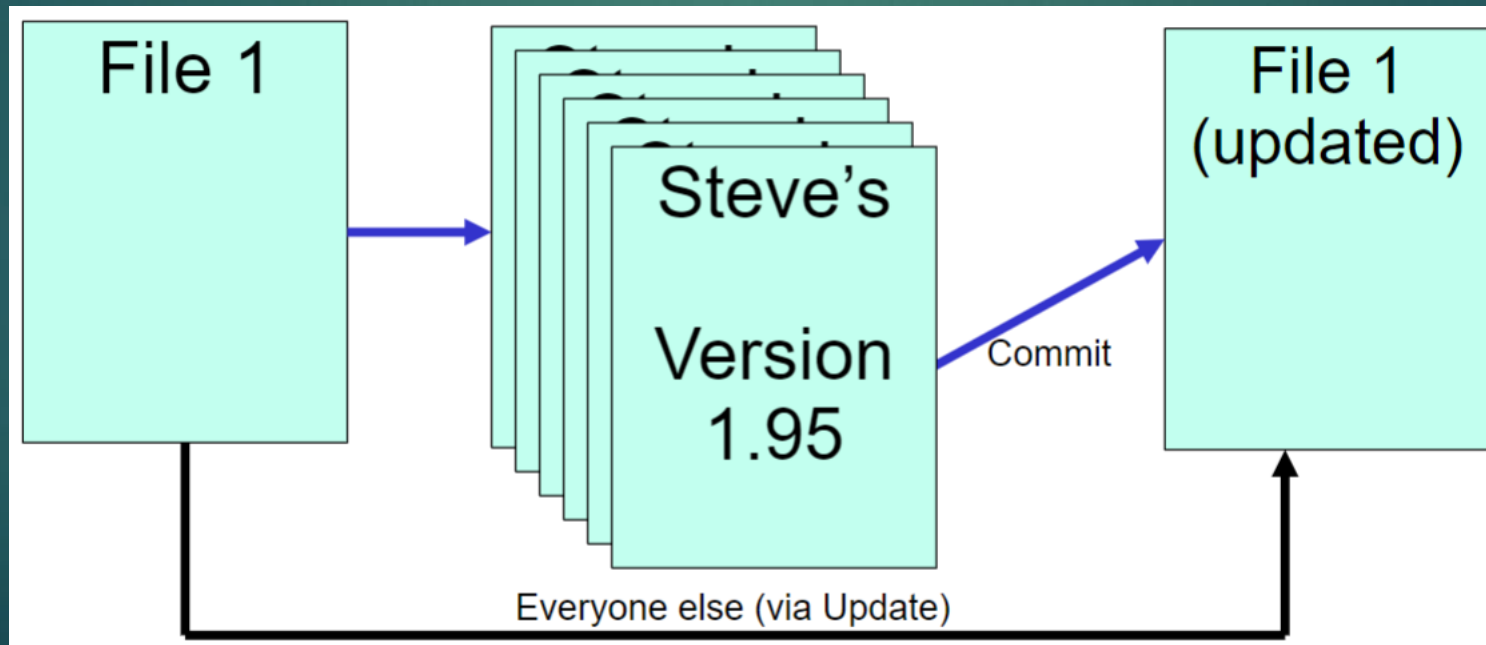


## Distributed (Git)



# Reason 1 Tracking Progress

- ▶ Allows one person to easily track their own progress
- ▶ If changes have caused problems, can be easily reverted



# Reason 2 for VC: Bugfixing

- ▶ You can “backtrack” commits to find where the bug was introduced

V. 127  
Bug  
Found

# Reason 1 for VC: Bugfixing

- ▶ You can “backtrack” commits to find where the bug was introduced

V. 126  
Bug  
Found

V. 127  
Bug  
Found

# Reason 2 for VC: Bugfixing

- ▶ You can “backtrack” commits to find where the bug was introduced

V. 125  
Bug  
Found

V. 126  
Bug  
Found

V. 127  
Bug  
Found

# Reason 2 for VC: Bugfixing

- ▶ You can “backtrack” commits to find where the bug was introduced

V. 124  
Bug not  
found

V. 125  
Bug  
Found

V. 126  
Bug  
Found

V. 127  
Bug  
Found

# Reason 2 for VC: Bugfixing

- ▶ You can “backtrack” commits to find where the bug was introduced

V. 124  
Bug not  
found

V. 125  
Bug  
Found

V. 126  
Bug  
Found

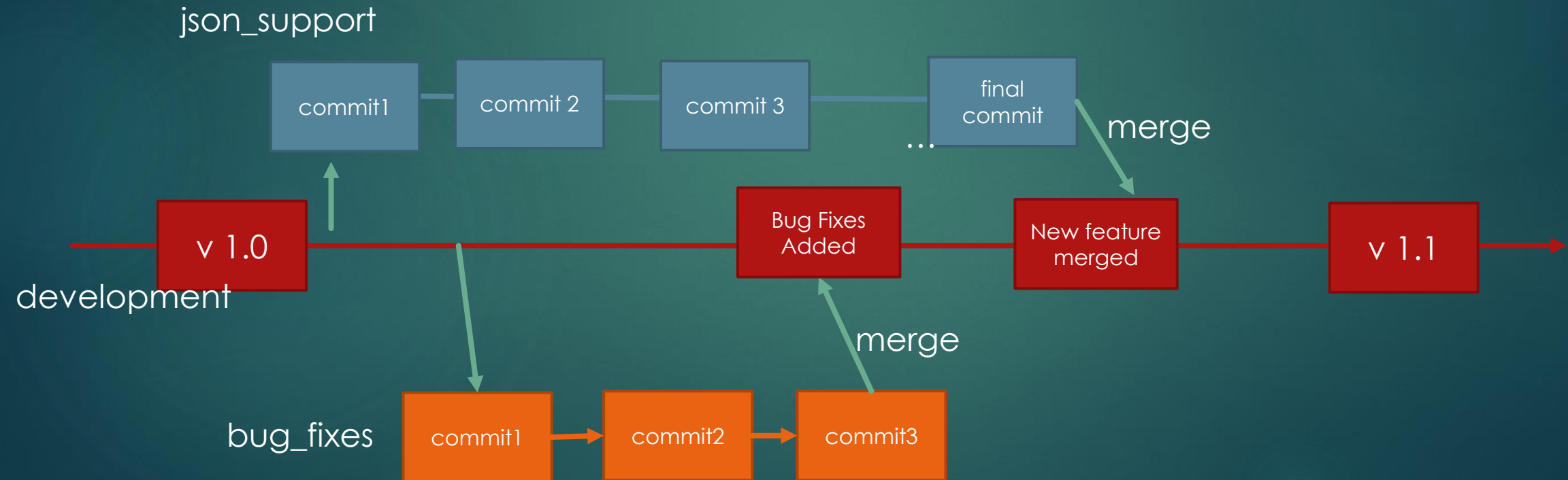
V. 127  
Bug  
Found

Bug likely has to do  
with this commit



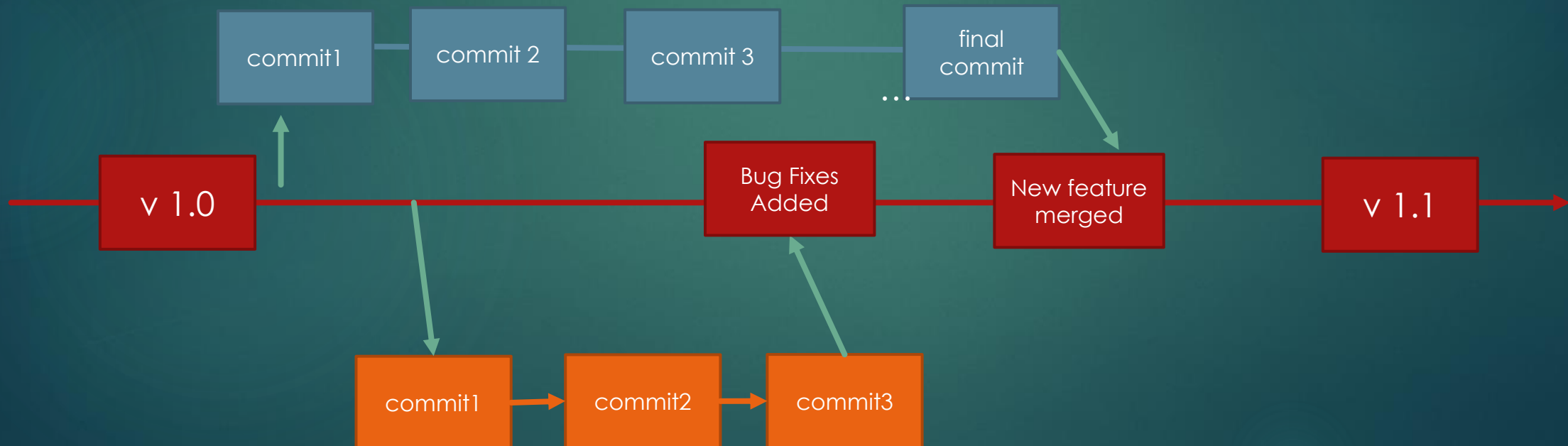
# Reason 3 for VC: Collaboration

- ▶ Multiple developers can simultaneously work together
  - ▶ Branching allows features to be added and developed independently



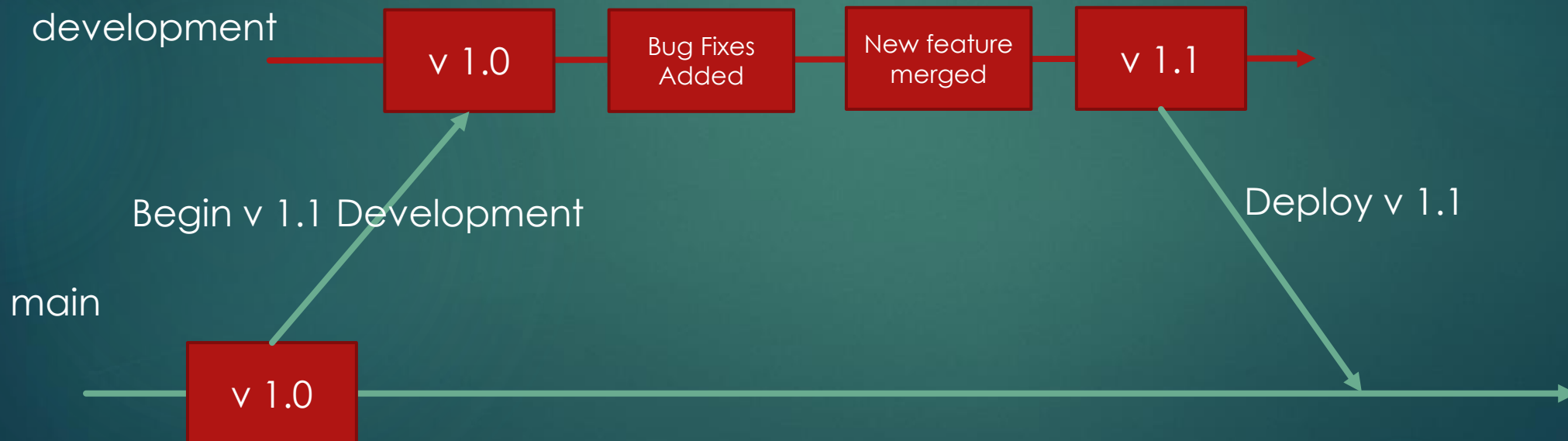
# Reason 3 for VC: Collaboration

- ▶ Each developer can be assigned to a feature
  - ▶ Each feature implemented independently in a separate branch
  - ▶ Everything merged to development branch

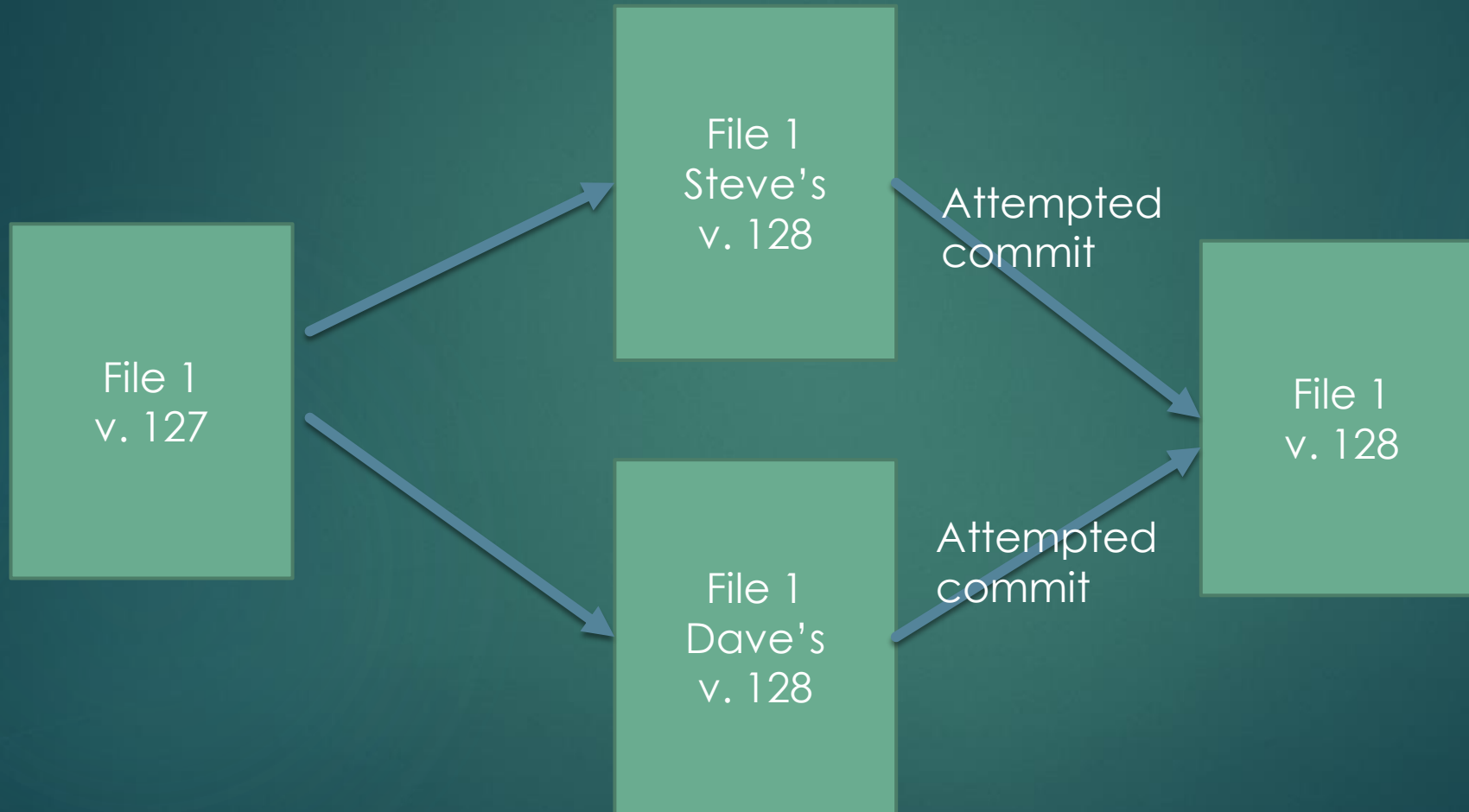


# Reason 4 for VC: Deployment

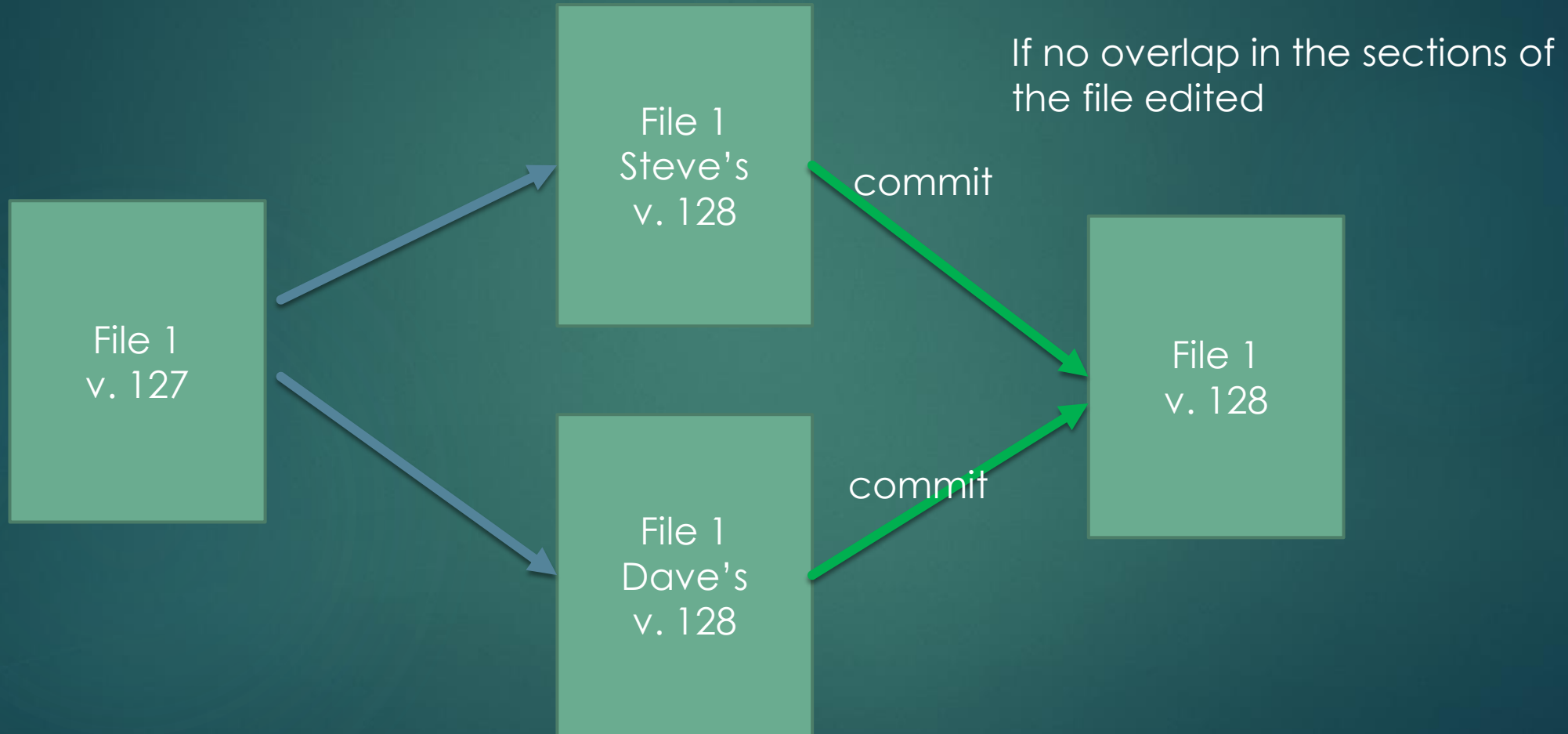
- ▶ The main branch acts as the currently released/available product
  - ▶ Changes are built in a separate development branch
  - ▶ Only merged when ready for release



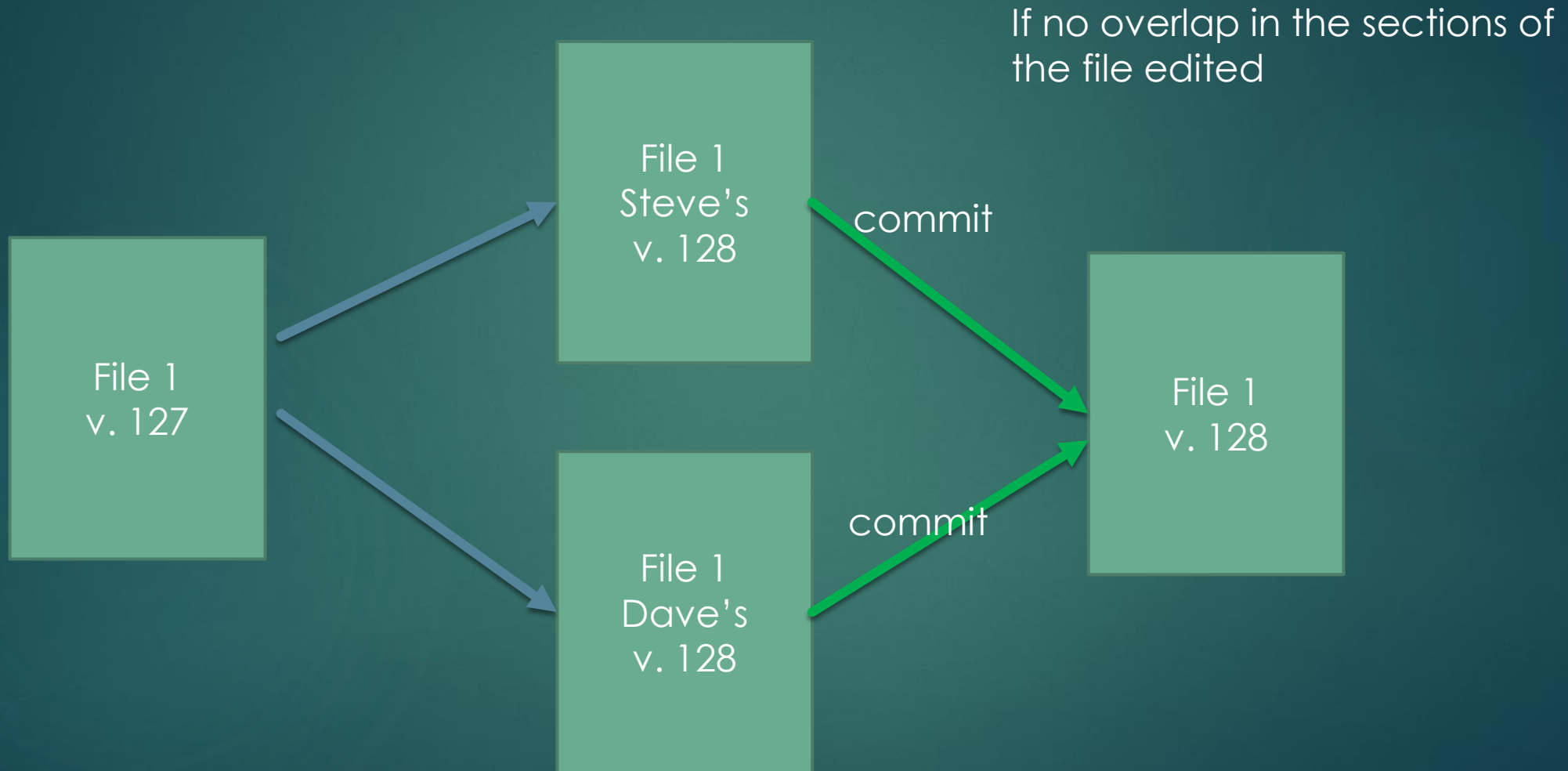
# Reason 5 for VC: Conflict Detection



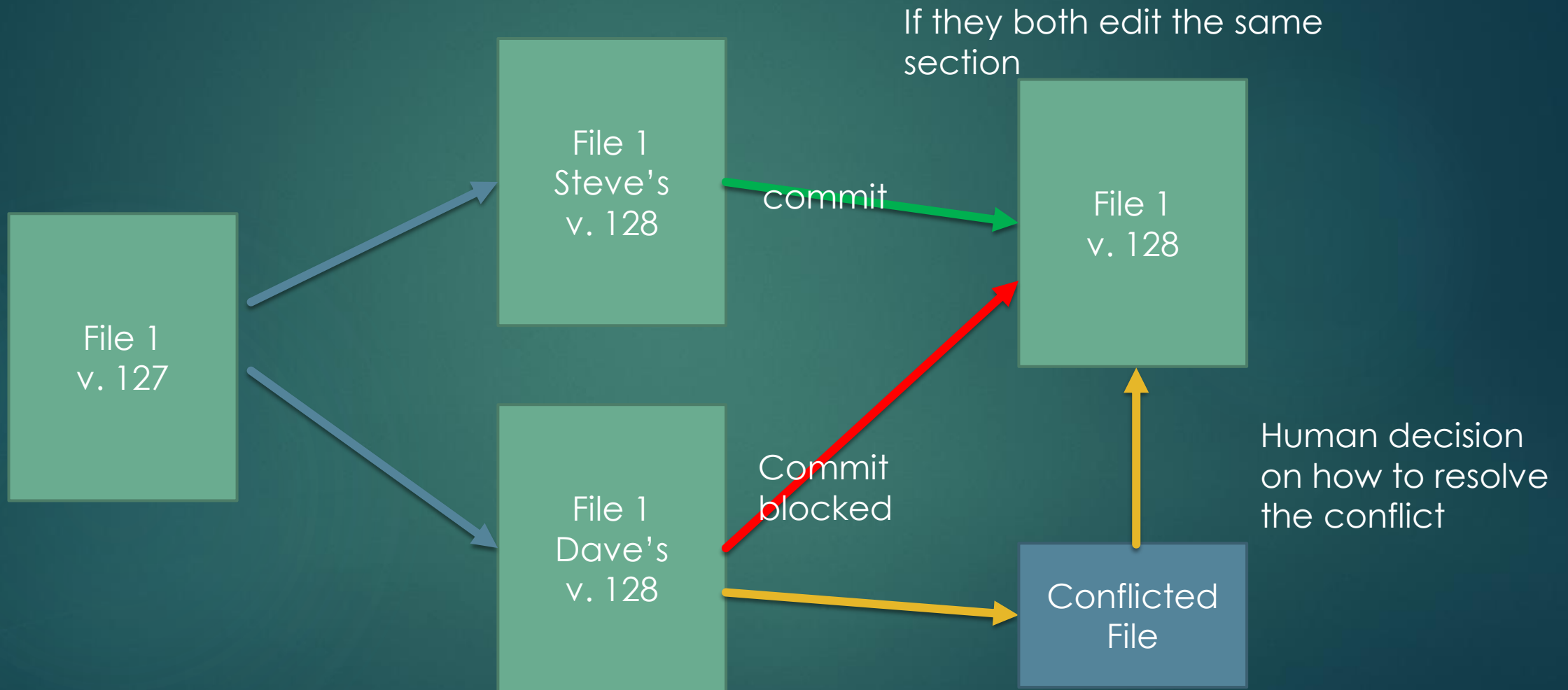
# Reason 5 for VC: Conflict Detection



# Reason 5 for VC: Conflict Detection

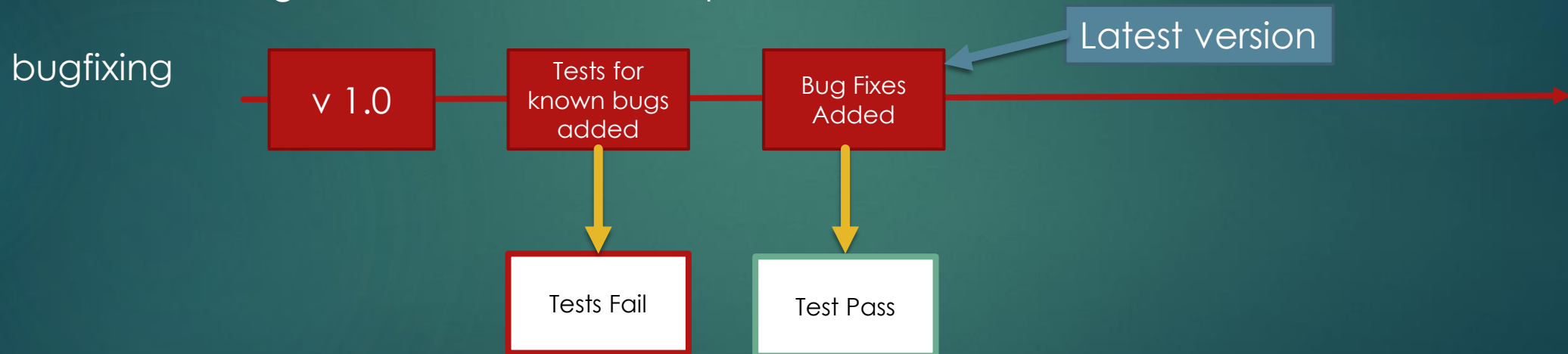


# Reason 5 for VC: Conflict Detection



# Reason 6 for VC: Regression Testing

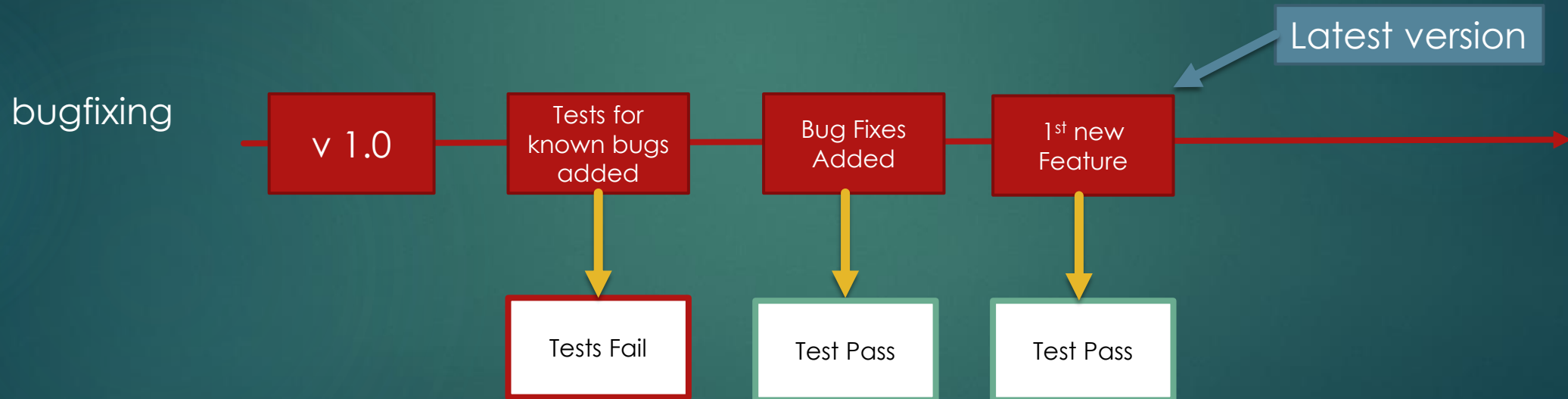
- ▶ Imagine as part of bug fixing, the bug fixing first added tests to ensure that the bug is actually fixed
  - ▶ Of course the tests fail at first (the bugs haven't been fixed yet) but the bug fixes make all the tests pass





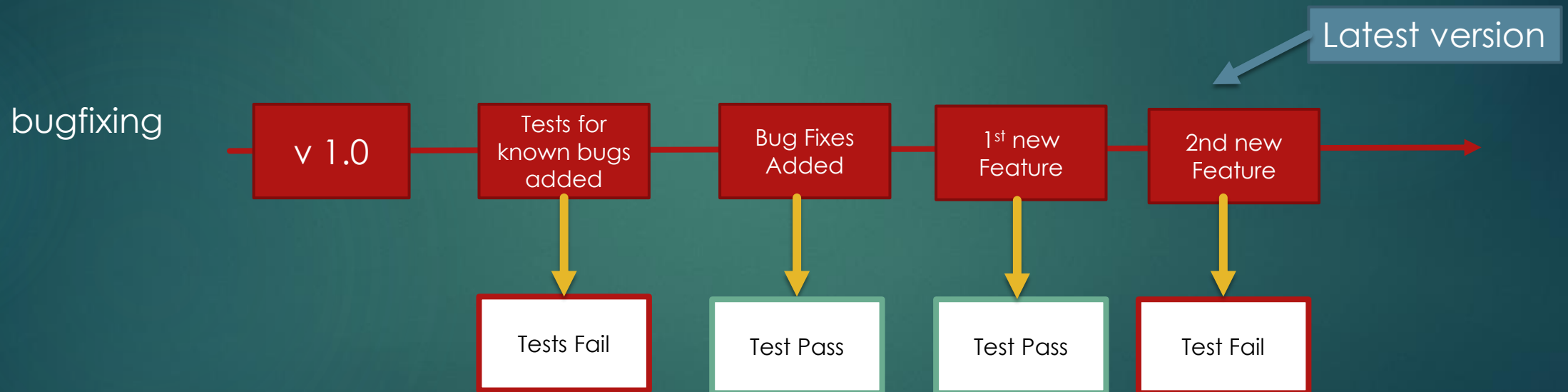
# Reason 6 for VC: Regression Testing

- ▶ Now you attempt to merge code from the “new feature branch”
  - ▶ You run the tests and they still pass, so you are confident the bugs are still fixed



# Reason 6 for VC: Regression Testing

- ▶ You try to merge a 2<sup>nd</sup> new feature
  - ▶ This times the tests fail!

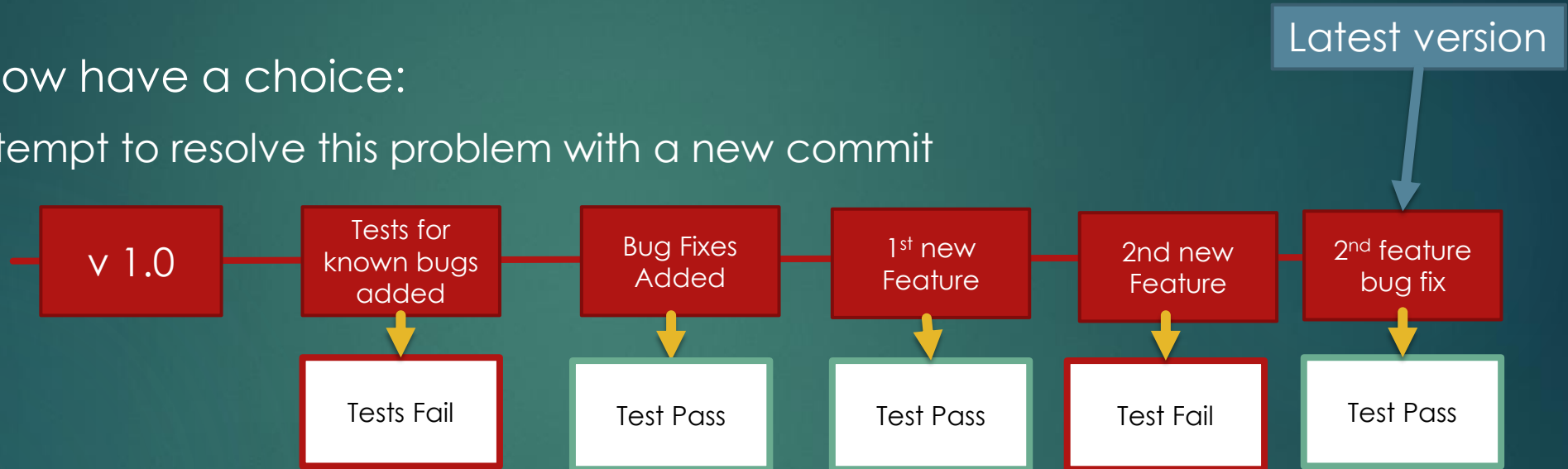


# Reason 6 for VC: Regression Testing

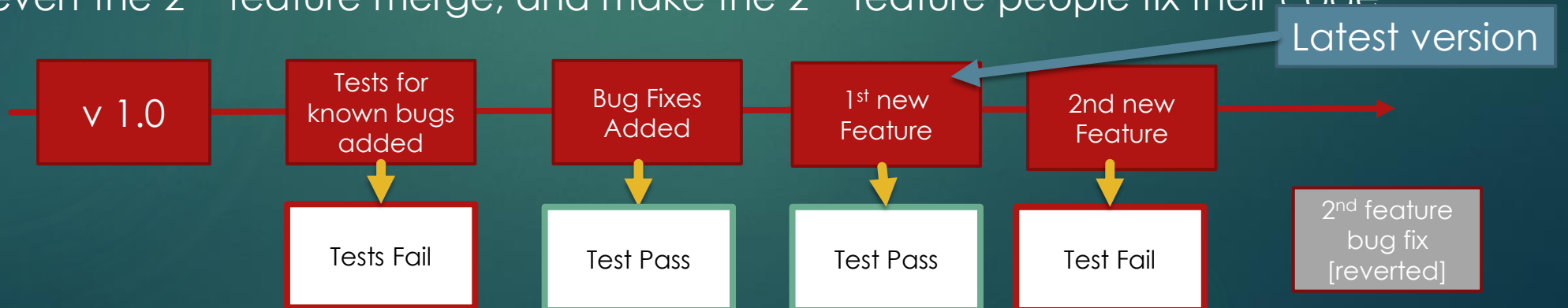
- ▶ You now have a choice:

- ▶ Attempt to resolve this problem with a new commit

bugfixing



- ▶ Revert the 2nd feature merge, and make the 2nd feature people fix their code



# Git Command Line



## Git Cheat Sheet

For more awesome cheat sheets  
visit [rebellabs.org](https://rebellabs.org)!



### Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

### Observe your Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

### Working with Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my\_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new\_branch

```
$ git branch new_branch
```

Delete the branch called my\_branch

```
$ git branch -d my_branch
```

Merge branch\_a into branch\_b

```
$ git checkout branch_b
```

```
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

### Make a change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

### Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

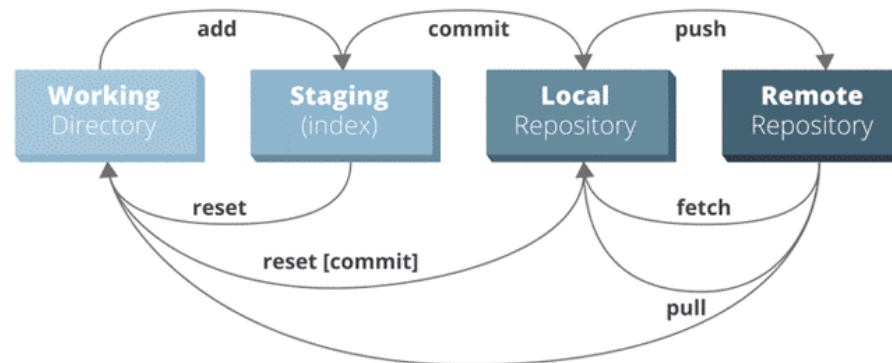
```
$ git push
```

### Finally!

When in doubt, use git help

```
$ git command --help
```

Or visit <https://training.github.com/> for official GitHub training.



# Commands to know

- ▶ init
- ▶ clone
- ▶ add
- ▶ commit
- ▶ push
- ▶ pull
- ▶ branch
- ▶ switch
- ▶ restore
- ▶ merge

# Git commands

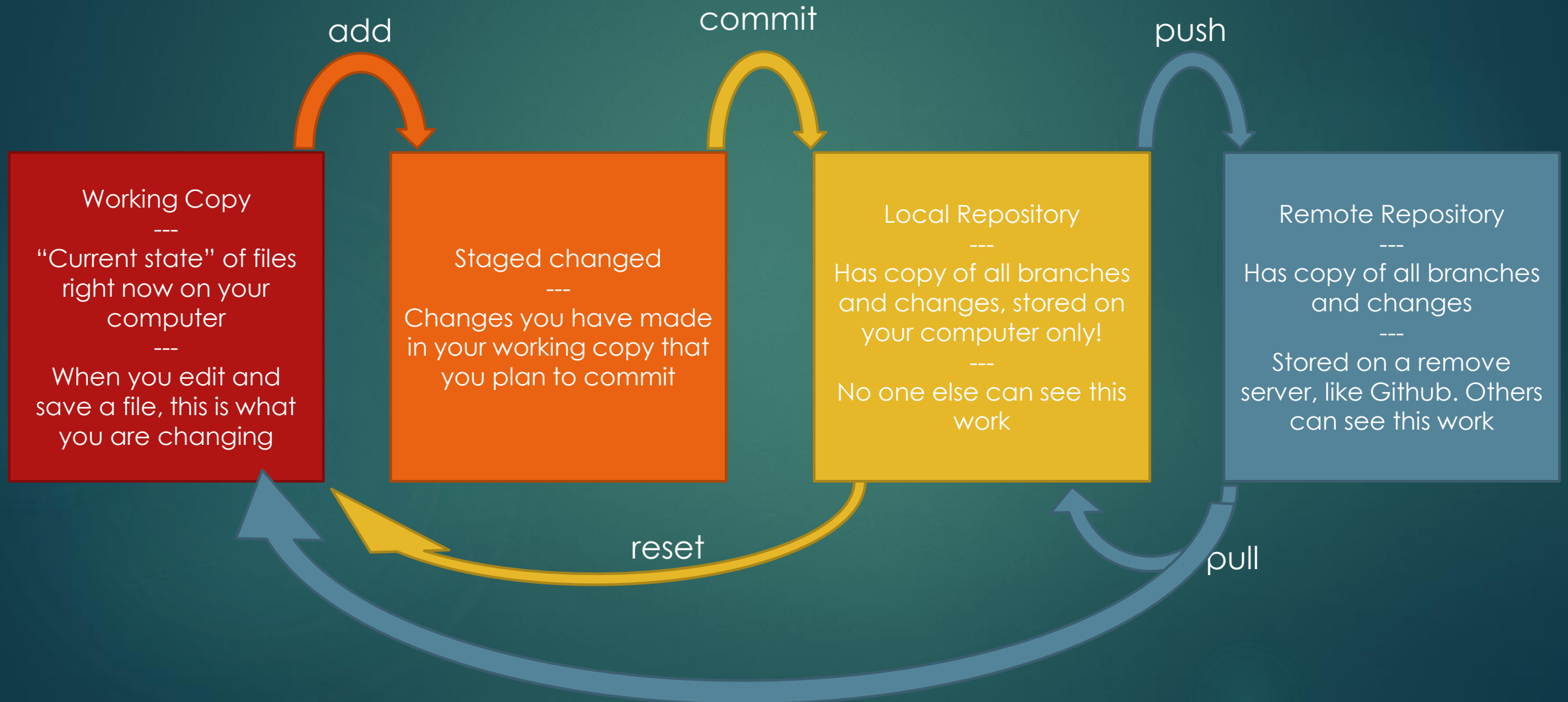
- ▶ git clone <https://github.com/sde-coursepack/packages.git>
  - ▶ Downloads the code in that directory to your current location
  - ▶ Puts everything in a folder called “packages” (the name of the repo)
- ▶ Use file structure to your advantage. Example file Structure
  - ▶ cs-3140
    - ▶ Homeworks (clone Homework repos to this folder)
      - ▶ hw1
      - ▶ Hw2
    - ▶ Class Notes (clone class notes repos here)

# Cloning directly into IntelliJ

- ▶ In IntelliJ, under File -> New...-> Project from Version Control
  - ▶ You can connect your Github account and directly clone your repositories
    - ▶ Under “GitHub”
  - ▶ “Repository URL” if you want to clone from a .git URL
- ▶ Cloning from Git
  - ▶ Use HTTPS for the simplest approach
    - ▶ You must use a [personal access token](#) as your password, not login password
  - ▶ If you have an ssh key setup, you can clone via ssh as well



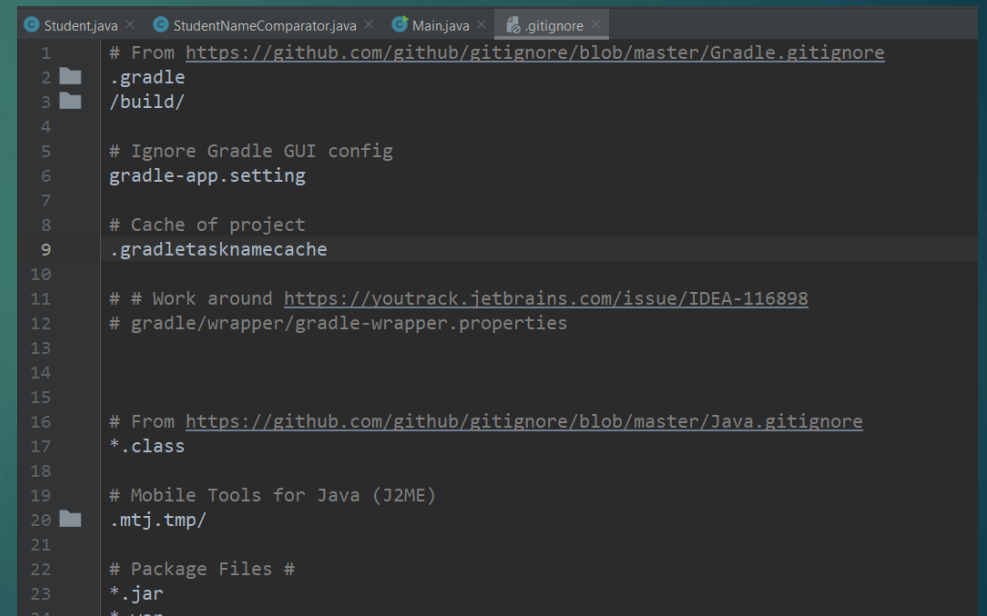
# Git add, commit, and push





# .gitignore

- ▶ In the root directory of your repo, you should always have a file named .gitignore
  - ▶ This lists all of the files you \*don't\* want to commit
  - ▶ We want to avoid committing files that
    - ▶ Are unique to our computer like .idea/workspace.xml
    - ▶ Built files like .class and .jar
    - ▶ Files contain private information like keys
- ▶ Example:
  - ▶ On the right, we say ignore
    - ▶ Build folder
    - ▶ .gradle folder
    - ▶ gradle-app.setting file
    - ▶ All .class files
    - ▶ All .jar files



```
1 # From https://github.com/github/gitignore/blob/master/Gradle.gitignore
2 .gradle
3 /build/
4
5 # Ignore Gradle GUI config
6 gradle-app.setting
7
8 # Cache of project
9 .gradletasknamecache
10
11 # # Work around https://youtrack.jetbrains.com/issue/IDEA-116898
12 # gradle/wrapper/gradle-wrapper.properties
13
14
15
16 # From https://github.com/github/gitignore/blob/master/Java.gitignore
17 *.class
18
19 # Mobile Tools for Java (J2ME)
20 .mtj.tmp/
21
22 # Package Files #
23 *.jar
24 *.war
```

# Git branches

- ▶ Your “default” branch is typically called main or master
  - ▶ If repo is created on Github it should be called main
- ▶ `git branch my_branch_name`
  - ▶ Create a new branch
  - ▶ Note, this doesn't \*change\* your branch!
- ▶ `git checkout my_branch_name`
  - ▶ Move to the branch named “my\_branch\_name”
- ▶ `git merge main`
  - ▶ Merge the code in main \*to your current branch\*.

# Git add, commit, and push

- ▶ If you want to push all current work in your current branch to github:
  - ▶ `git add .`
    - ▶ The period is a “wildcard”, meaning all files (respects .gitignore)
  - ▶ `git commit -m “A meaningful message”`
    - ▶ In later assignments, you will be graded on commit message quality
  - ▶ `git push`
    - ▶ Pushes all commits since your last push to github
    - ▶ You don’t have to push after every commit!

# Best practices

- ▶ 1) Never do new work in main branch
  - ▶ If everyone is working in main, you're going to have tons of conflicts
- ▶ 2) Also Pull before pushing
- ▶ 3) When merging, merge from main to branch FIRST
  - ▶ Handle conflicts in the branch
  - ▶ Only merge back to main when conflicts are resolved and tests passing
- ▶ 4) "Commit early and often"
  - ▶ Notice it doesn't say "push early and often"
  - ▶ You should commit after each "unit" of work (function, test, bug fix, etc.)
- ▶ 5) Push when you are ready to share
  - ▶ If you are stopping for the day, it's good to push so your changes are up to date
- ▶ While you won't be graded on git practice for Homework 1, you will be graded on it as the semester progresses!

# Do we need have to use command-line

- ▶ No, it is completely fine to use a desktop application
  - ▶ IntelliJ has built-in git support that is pretty solid
    - ▶ (I use this)
  - ▶ There are a ton of GUI applications
    - ▶ GitHub Desktop
    - ▶ GitKraken
    - ▶ Tortoise Git
    - ▶ SourceTree
  - ▶ Pick something that works for you!
- ▶ It's worth knowing the basics of command line, however

# Best workflow practice - Starting

- ▶ Before you start writing code
- ▶ `git pull`
  - ▶ Get all the latest changes since you last worked
- ▶ `git branch name_of_your_branch`
  - ▶ Create the branch you will work in if you are making a new branch
  - ▶ If working in existing branch, checkout
- ▶ `git switch name_of_your_branch`
  - ▶ Move to that branch (checkout doesn't move you)
- ▶ Now you can start coding!

# Best workflow practice - Uploading

- ▶ After you are done writing code in your branch and reading to share it to Github:
- ▶ `git add .`
- ▶ `git commit -m "what you've done since last commit"`
- ▶ `git pull`
  - ▶ Always pull before pushing, that way if anyone else has pushed, you can resolve conflicts as early as possible
  - ▶ Smaller conflicts are far easier to resolve
- ▶ `git add .`
- ▶ `git commit -m "merged changes with remote"`
- ▶ `git push`



# Best workflow practice - Merging

Make sure your changes are pushed to your branch

- ▶ `git checkout main`
- ▶ `git pull`
  - ▶ Go to the main branch and ensure you have the up to date changes
- ▶ `git switch your_branch_name`
  - ▶ Go back to your branch
- ▶ `git merge main`
  - ▶ Handle the merge in your own branch *\*first\** - all conflicts should be resolve here
- ▶ `git add/commit/push`
- ▶ `git switch main`
- ▶ `git merge my_branch_name`
  - ▶ Now merge your branch into main



# If you can't pull (telling you to “stash changes”)

- ▶ git stash
  - ▶ Store any changes you made for later
  - ▶ Resets your working copy to last commit
- ▶ git pull
  - ▶ Pull in the remote changes
- ▶ git stash pop
  - ▶ You will likely have conflicts, so handle them now
- ▶ Add/Commit/Push
  - ▶ Send your merge to the branch