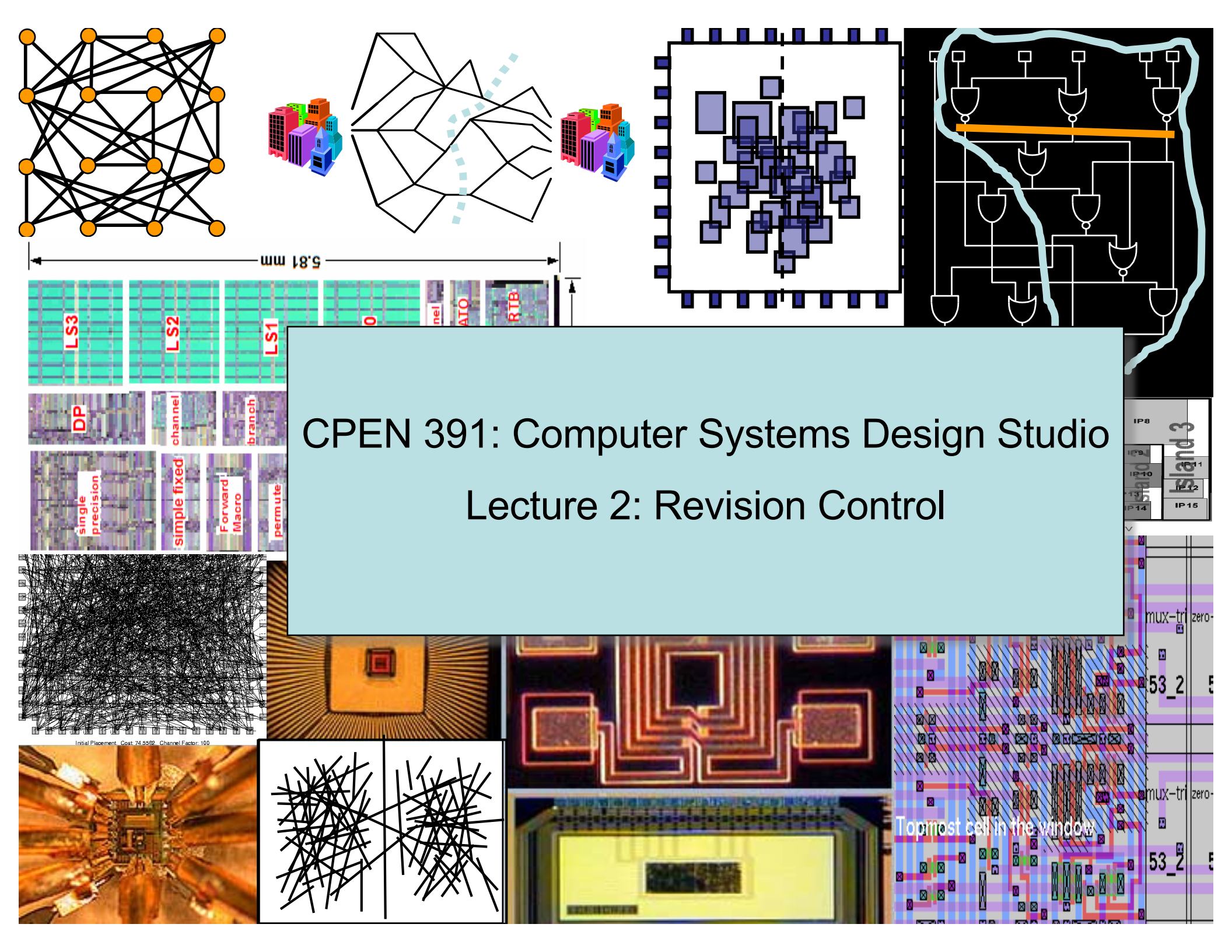



# CPEN 391: Computer Systems Design Studio

## Lecture 2: Revision Control

# Managing Your Code

- The same thing…
  - Software Configuration Management
  - Revision Control
  - Version Control

- But what is it?
  - A practice for keeping track of changes to your source files
  - Tools to help you

- Who should do it?
  - Everyone!
  - Individual developers
  - Teams of developers
  - Hardware or software

# Revision Control Systems – Tools

- Old guard
  - SCCS, RCS, DCS, PVCS, DDE, Perforce …
- Microsoft
  - Visual SourceSafe (VSS) (deprecated)
  - Team Foundation Version Control (TFVC) or Git
- Open source favourites: CVS, Subversion, Mercurial
- The Rolls Royce: IBM Rational ClearCase($$$)
- New darling: GIT
  - Used by Linux kernel
  - Intended for wide-scale distributed development
  - Also useful for individuals and teams

# RCS Tools – which to use?

- There are ~40 different RCS tools
    - List

http://en.wikipedia.org/wiki/List_of_version-control_software

    - Comparison

http://en.wikipedia.org/wiki/Comparison_of_revision_control_software

- This course: GIT
    - Most popular among CompEng profs in ECE
    - Used in my company
    - In particular, http://github.com is recommended
        - Free accounts for students
        - Free accounts for open source projects

# RCS – simple revision control

RCS comes with Linux and (all) versions of Unix

- Often used to remember changes to “Linux config files”

- Save a copy of the current file, as a revision:

```
$ ci dhcpd.conf
```

  - This is stored in file named dhcpd.conf,v

- Check out a read-only copy of the file for use:

```
$ co dhcpd.conf
```

- Check out a writable copy, or make read-only copy writable:

```
$ co -l dhcpd.conf      (dash-lowercase-L for “lock”)
```

(modify file)

```
$ ci dhcpd.conf         (save new version)
```

# GIT – Basics

- Command-line interface (standardized)
  - Many GUI tools, not really standardized
- Use under “Linux” or “Cygwin on Windows”
- Eg, install it as part of Cygwin:
  - Go to cygwin.com, download setup-x86.exe or setup-x86_64.exe
  - Start → Altera → Nios → Nios Command Shell
  - cd /cygdrive/c/users/myusername/downloads
  - ./setup-x86_64.exe
  - Install under c:\cygwin64
  - Select git, git-completion, git-gui, gitk packages
  - Install them (and any dependencies)

# Git

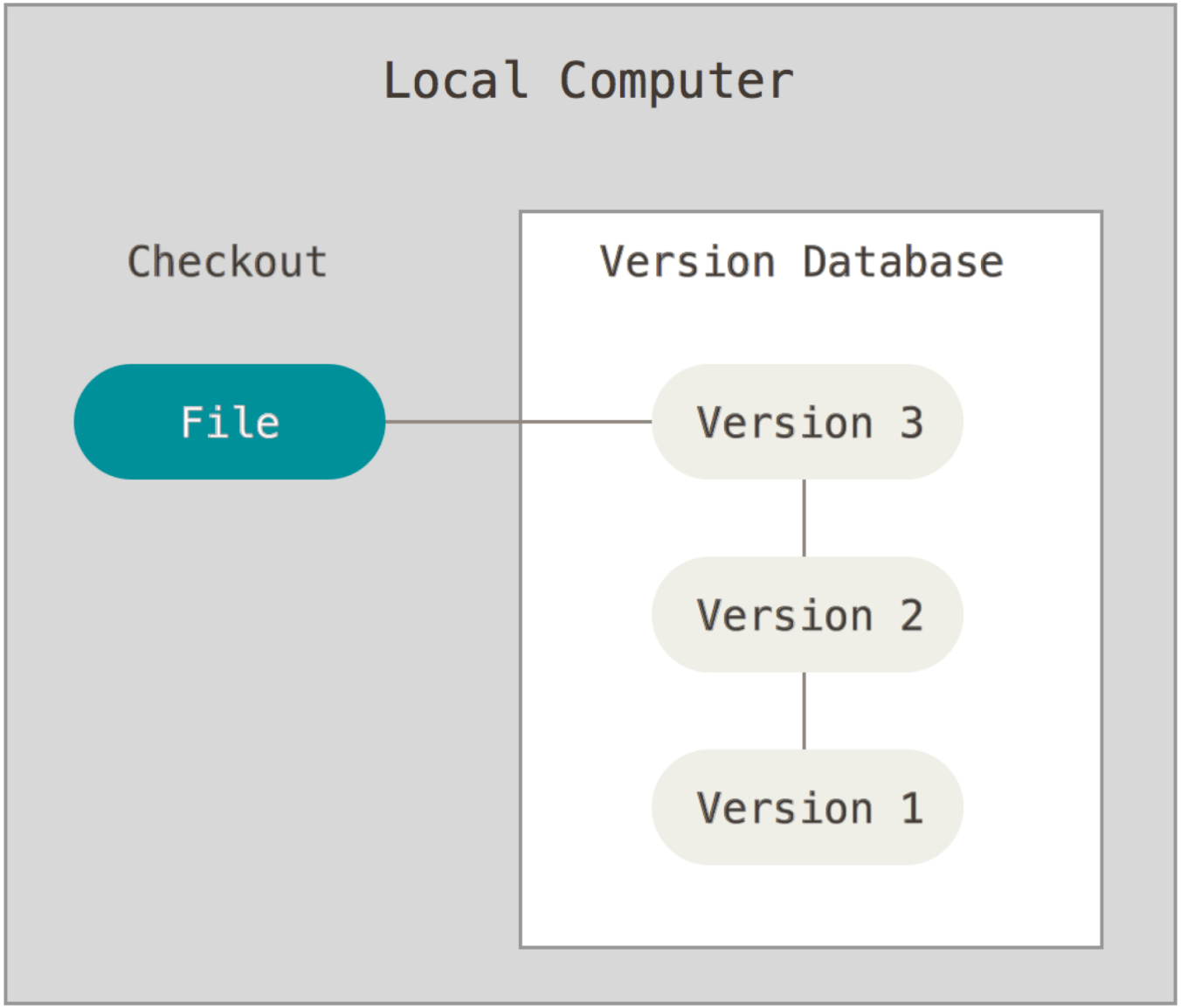
- Git works with “repositories”
  - It is decentralized by nature
  - But you can use a central repository (best for small teams)

- Great GIT tutorials
  - http://www.atlassian.com/git/tutorials
  - Eg: “Using Branches” and “Comparing Workflows”
  - Aside: Atlassian runs BitBucket, which is similar to GitHub

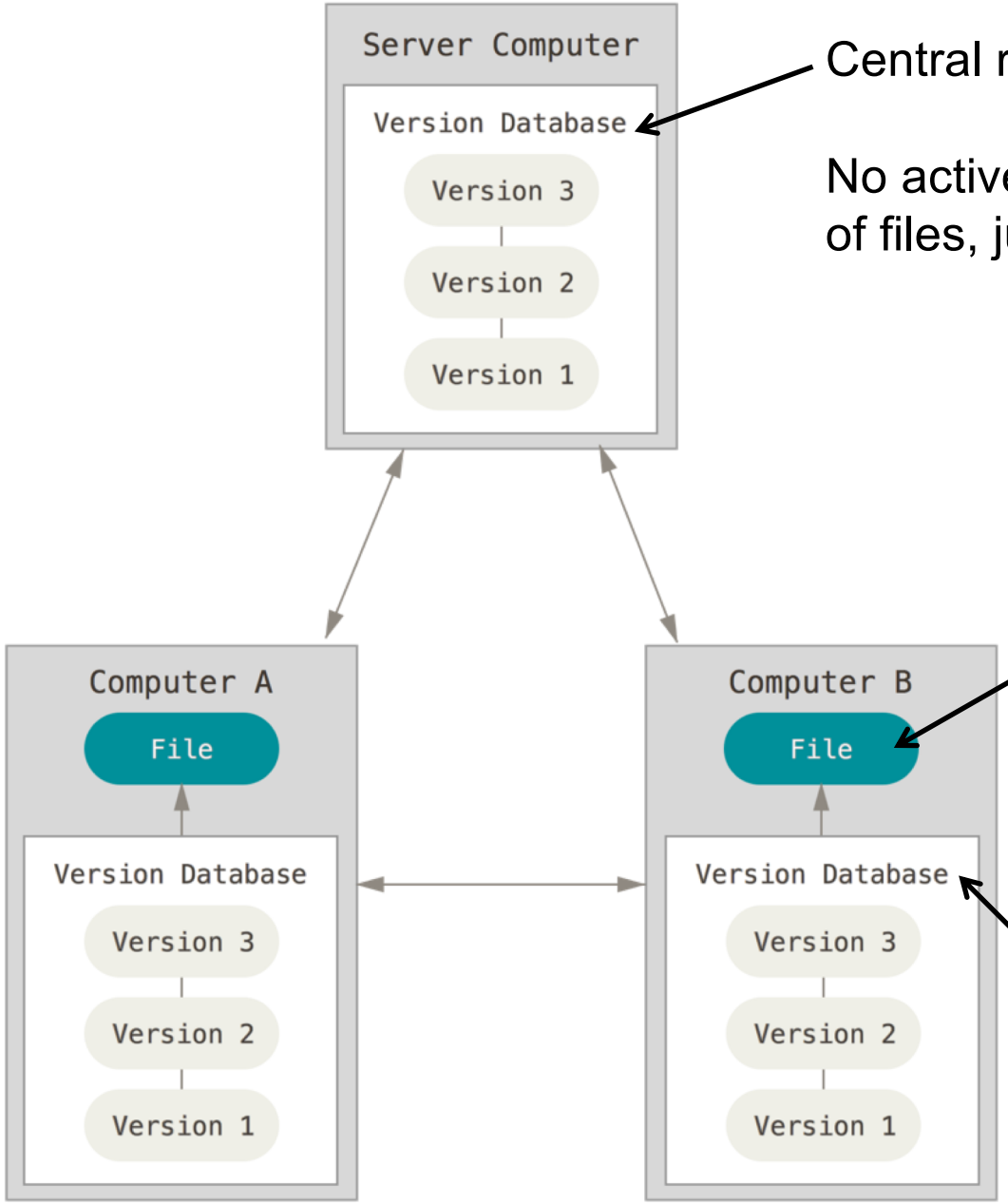
- Official documentation
  - http://git-scm.com/doc

This lecture will borrow figures and content from both of these sites, and more …

# Git – Main Idea



# Git – Decentralized Control




Central repository, eg: github

No active “working copies” of files, just stored versions

Your working file set (edit, compile, run)

Your personal repository/history

# Creating Git Repositories

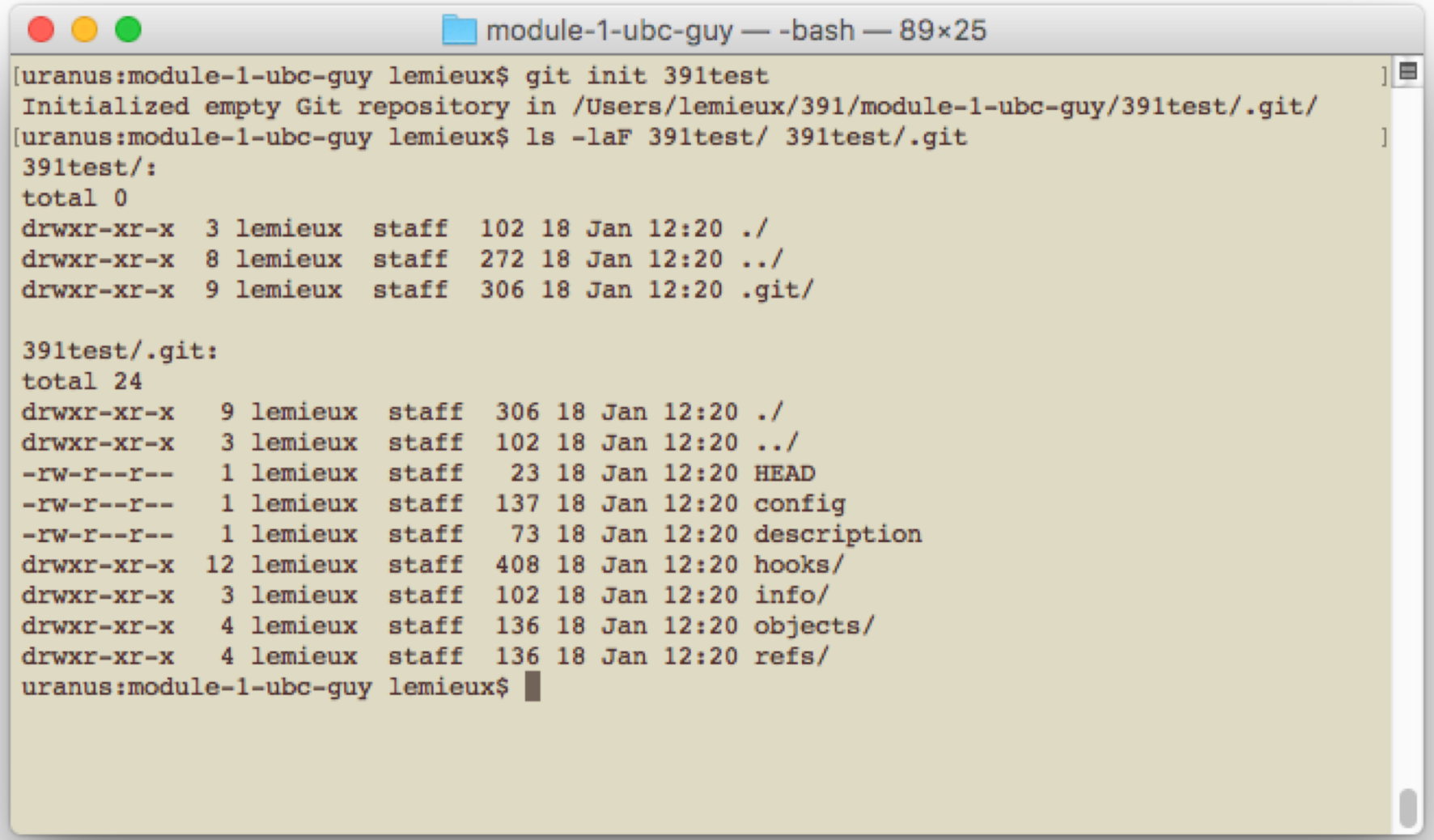
Cover 2 ways to create git repositories

1. Local respository (on PC, personal development only)
2. Remote repository (in cloud, personal or team-based)

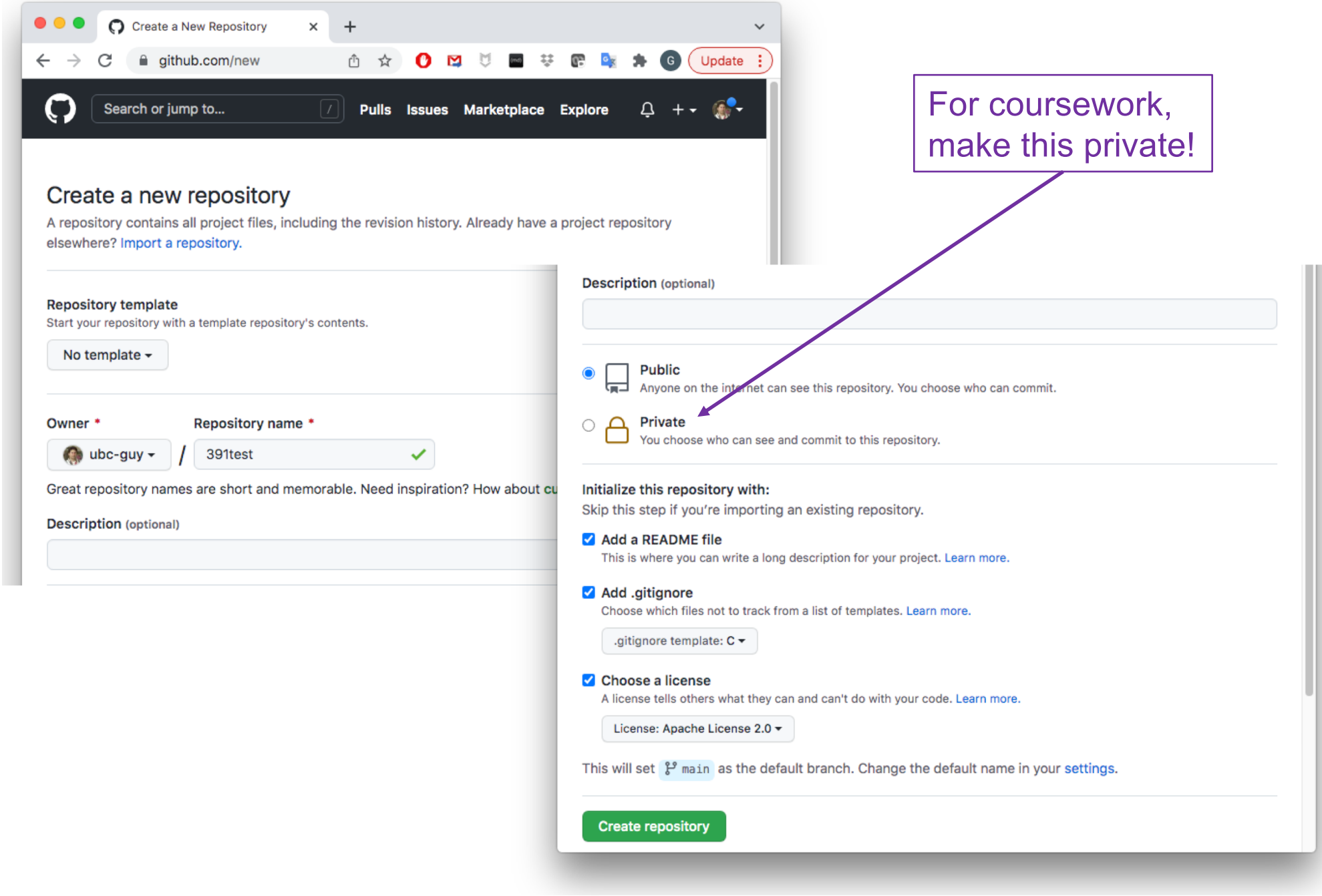
# Git – Getting Started (Create Local Repository)

- Private local repository on your computer:

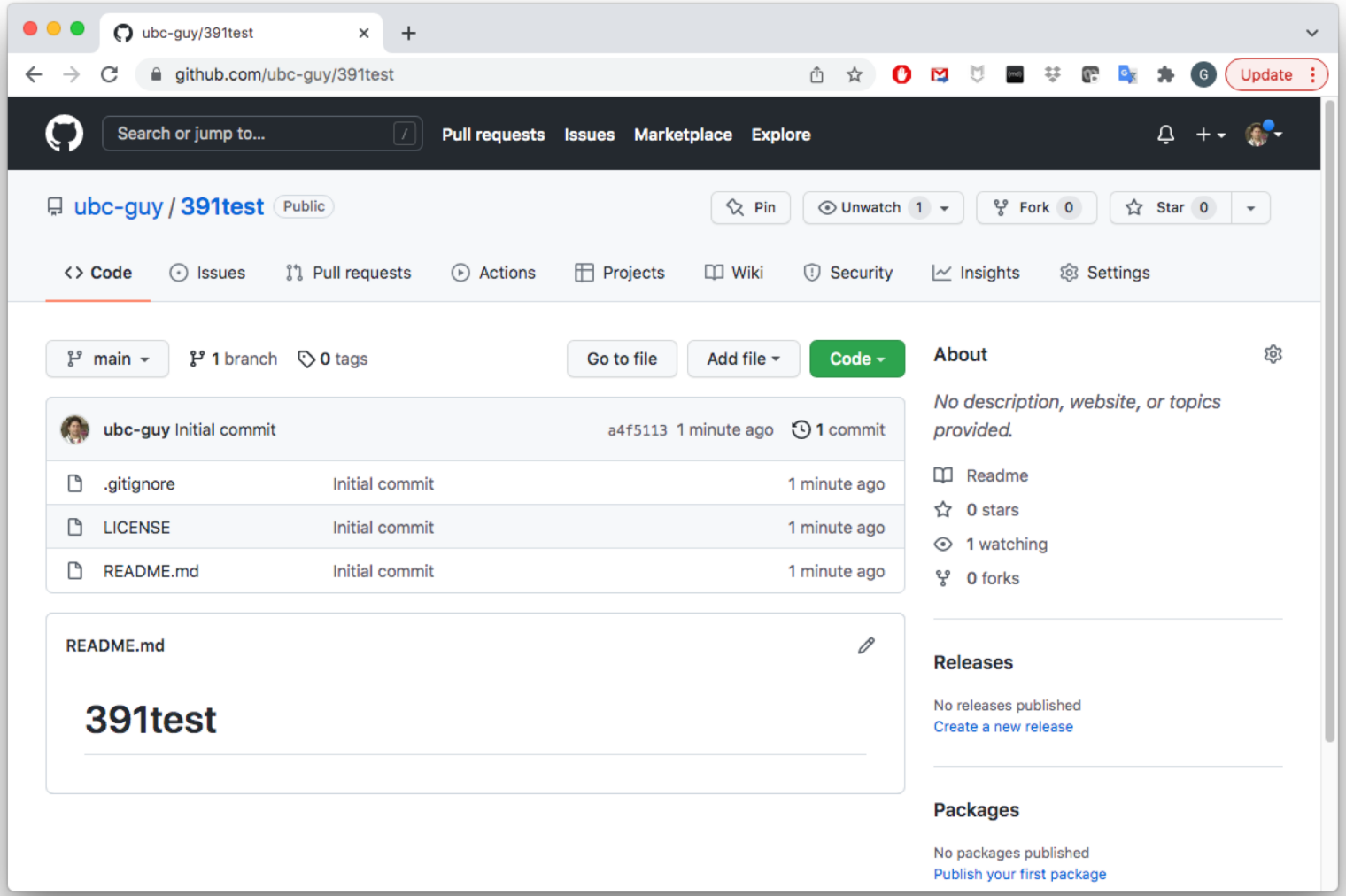
```
$ git init 391test
```



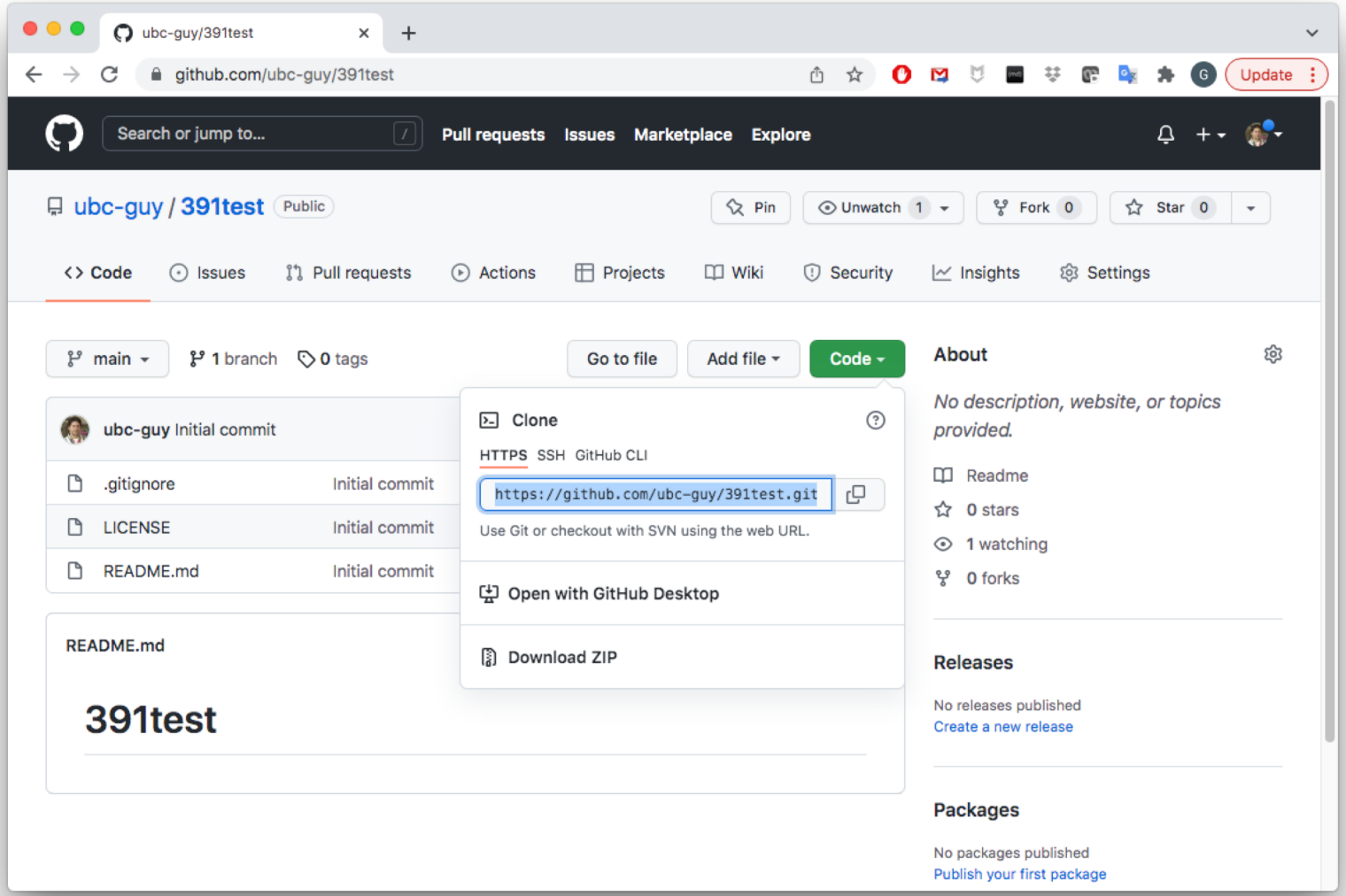
# Git – Getting Started (Create Remote Repository)



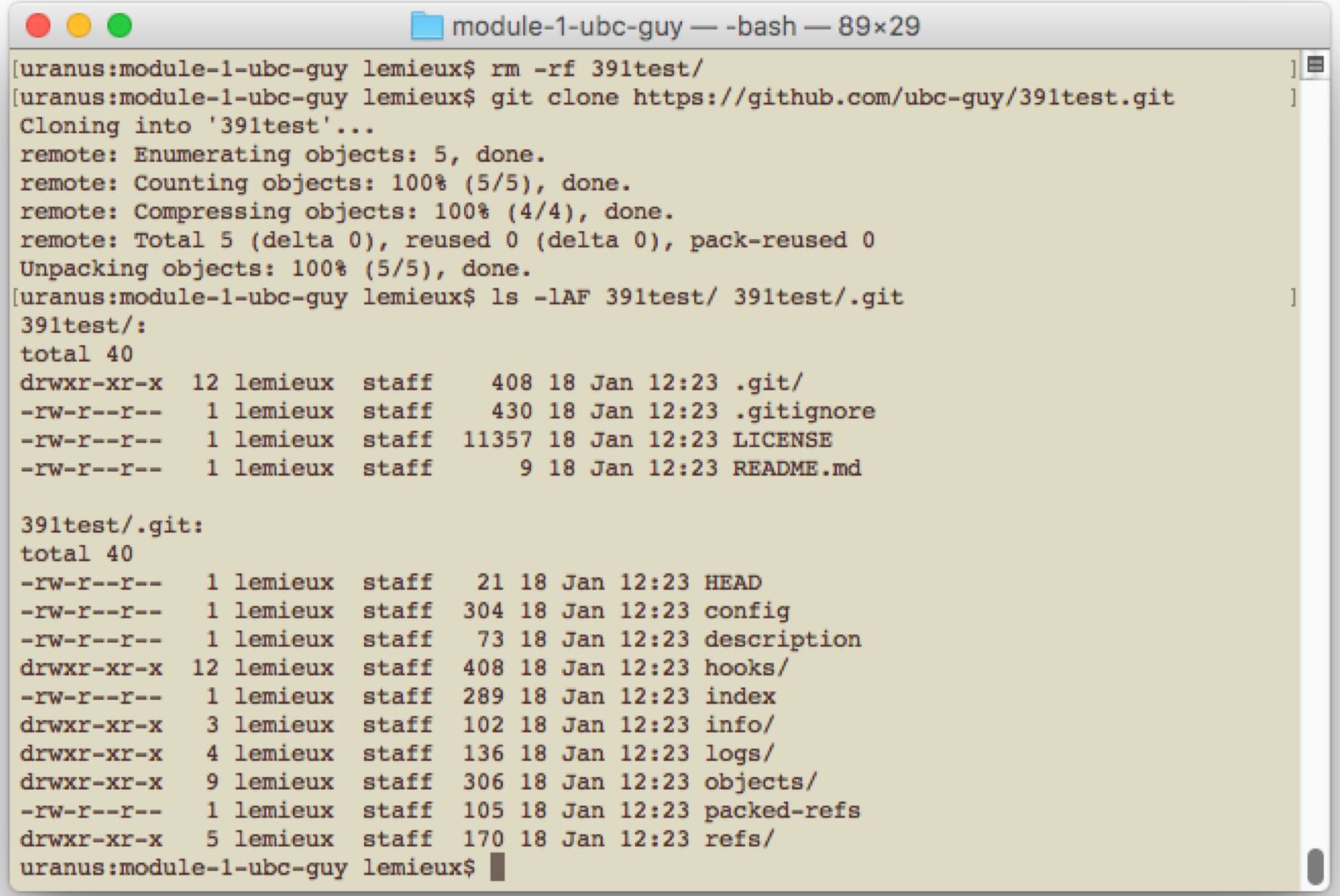
# Git – Getting Started (Remote Repository Result)



# Git – Getting Started (URL to Remote Repository)

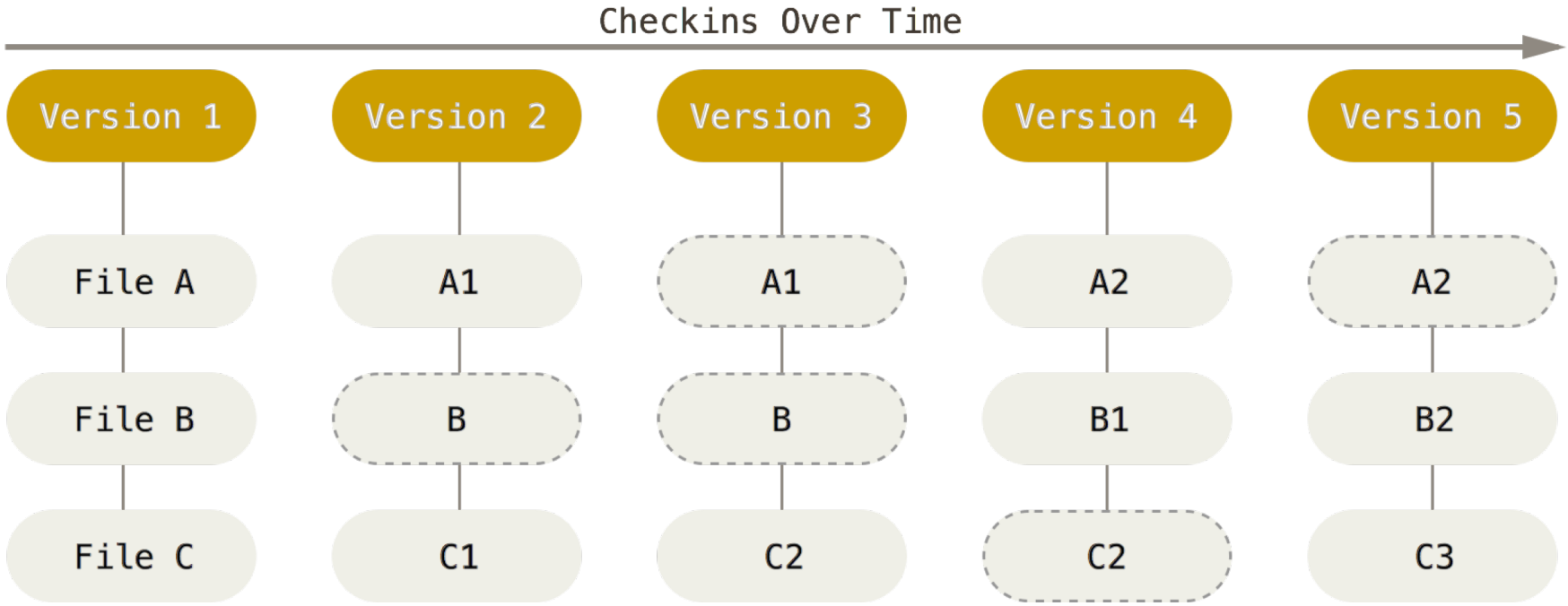


# Git – Getting Started (Local Clone of Remote Repository)



# Git Files – Checking In

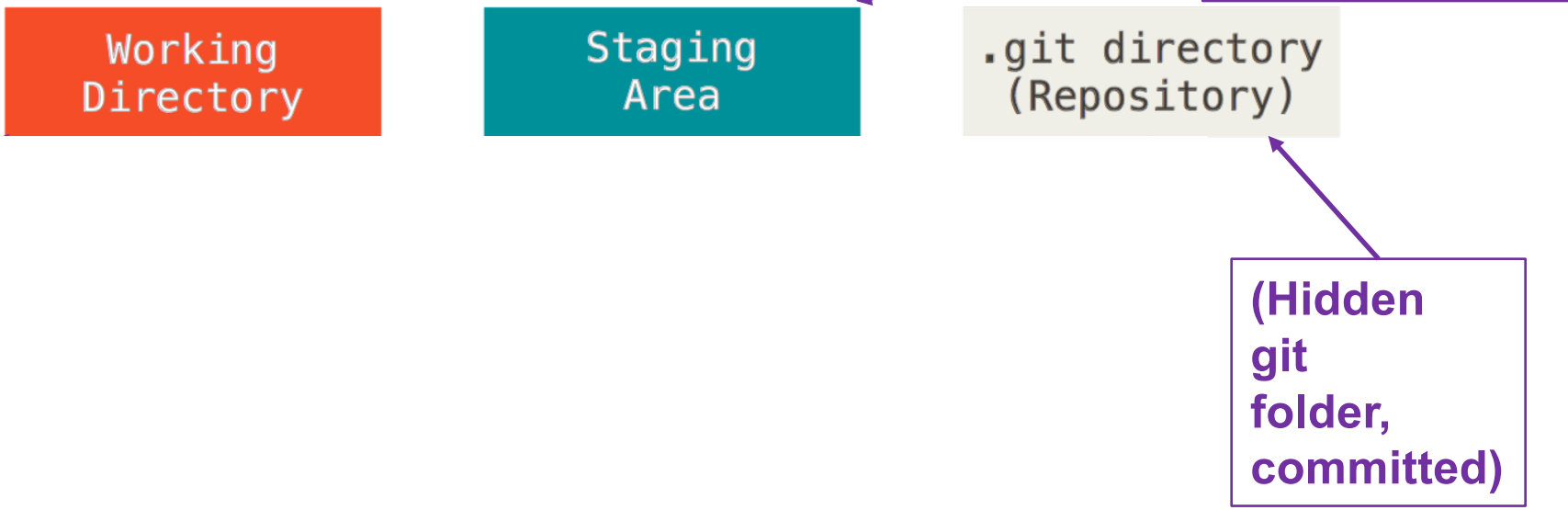
- Git stores snapshots of your file set, saved over time




- Nearly every operation is local (does not involve a remote server)
- Every version is stored and named using a SHA-1 hash, eg:
  `24b9da6552252987aa493b52f8696cd6d3b00373`
- Git usually only adds data
  - Don't panic, almost any operation can be "undone"

# Git – Staging Area

- Two stages: “add” followed by “commit”
- Add allows you to build one **consistent** set of files into staging area
  - Most of time, one feature/fix changes several files at once
- Commit when you are done adding



# Basic Git Workflow

The basic Git workflow goes something like this:

1. You modify files in your working directory.

```
(modify file1.c, file2.c, file.h)
```

2. You stage the files, using “git add”, which puts a snapshot of each added file in the staging area.

```
$ git add file1.c file2.c file.h
(modify primes.c)
$ git add primes.c
```

3. You do a commit, using “git commit” which takes the files from the staging area and stores that permanently in your Git repository

```
$ git commit
```

# Git – Your first commit

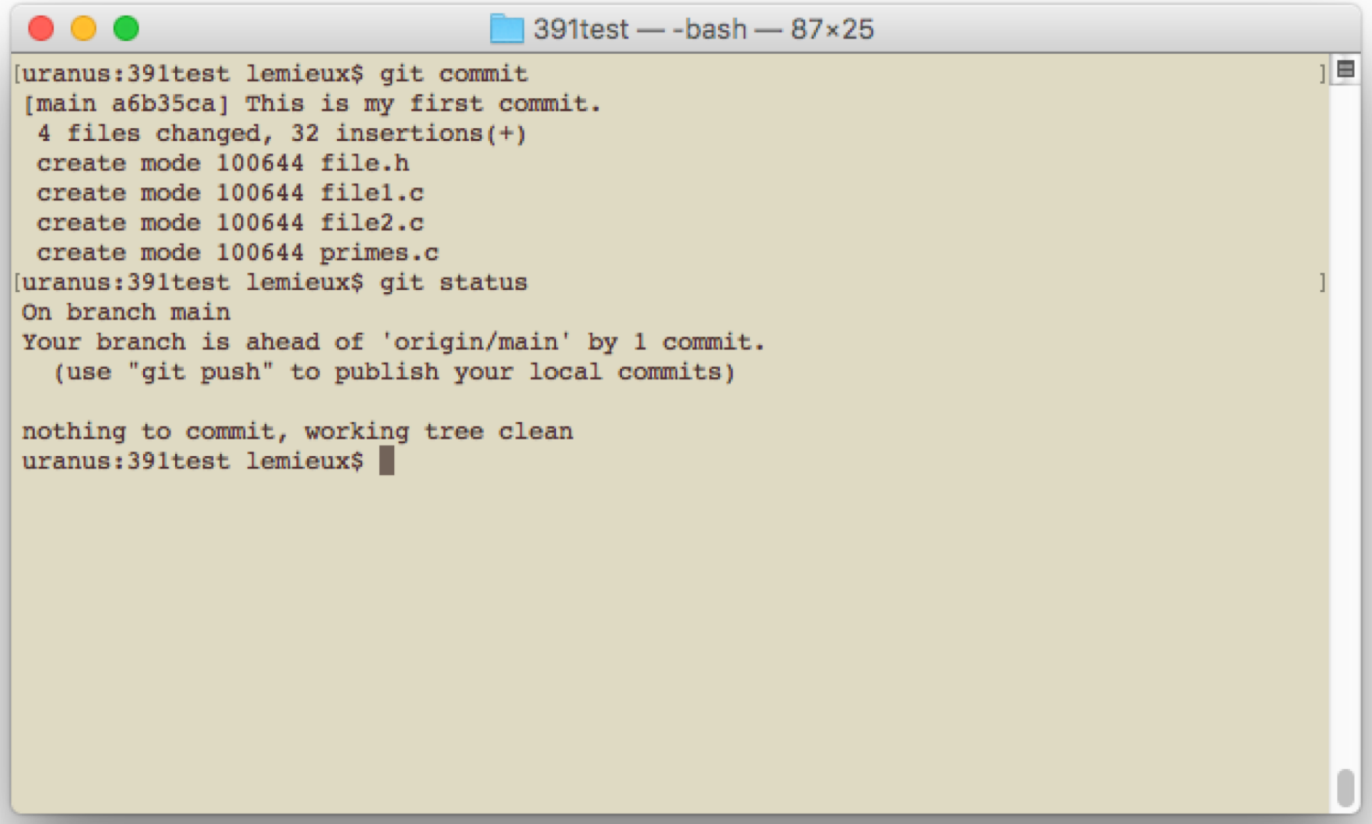
# Git Commit – interactive editor

# Git Commit – interactive editor

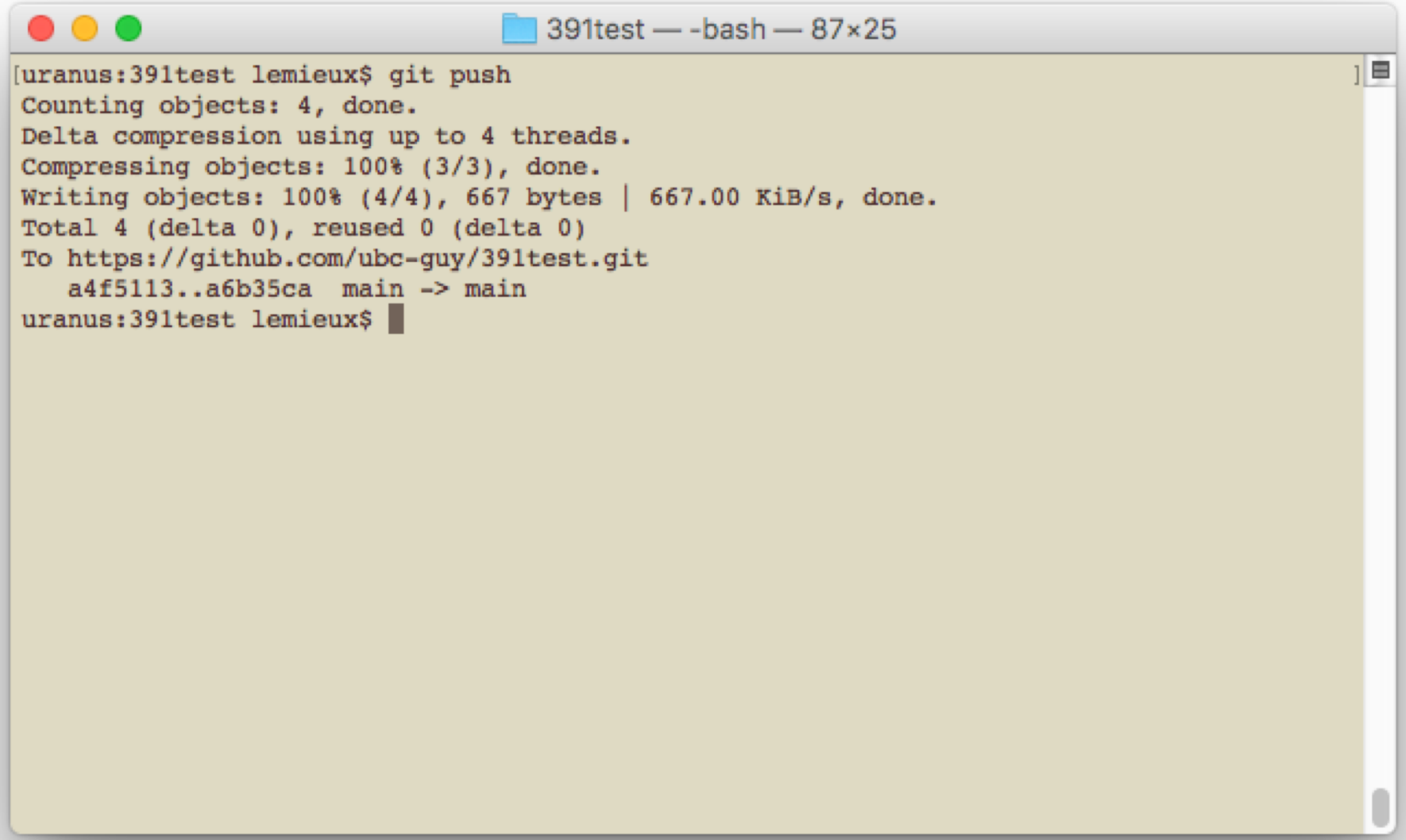
# Git Commit – status after commit

- Your changes are in your local PC repository
  - They have not been sent to the “shared repository”

- At any time, you can run “git status” to receive a short report on what has changed from the repository:



# Git – “push” to central repository

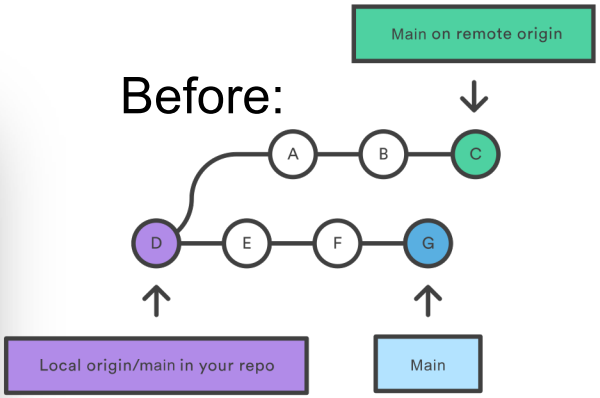


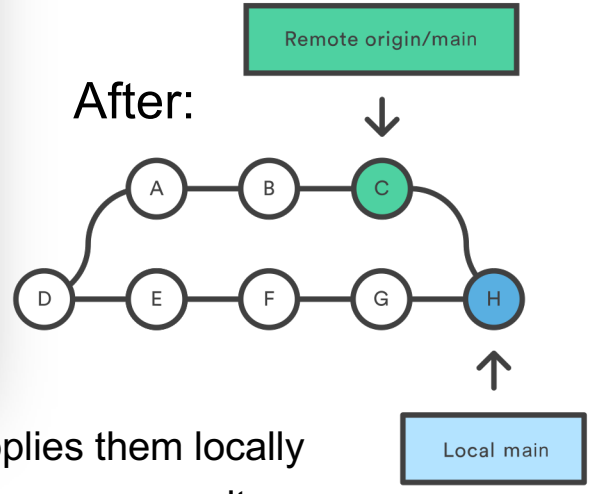
# Git – Collaborations

- Suppose my team-mate has modified primes.c and file.h
  - They correctly did "`git add`", "`git commit`", and "`git push`"
  - Retrieve all changes using "`git pull`"

391test — -bash — 87×25

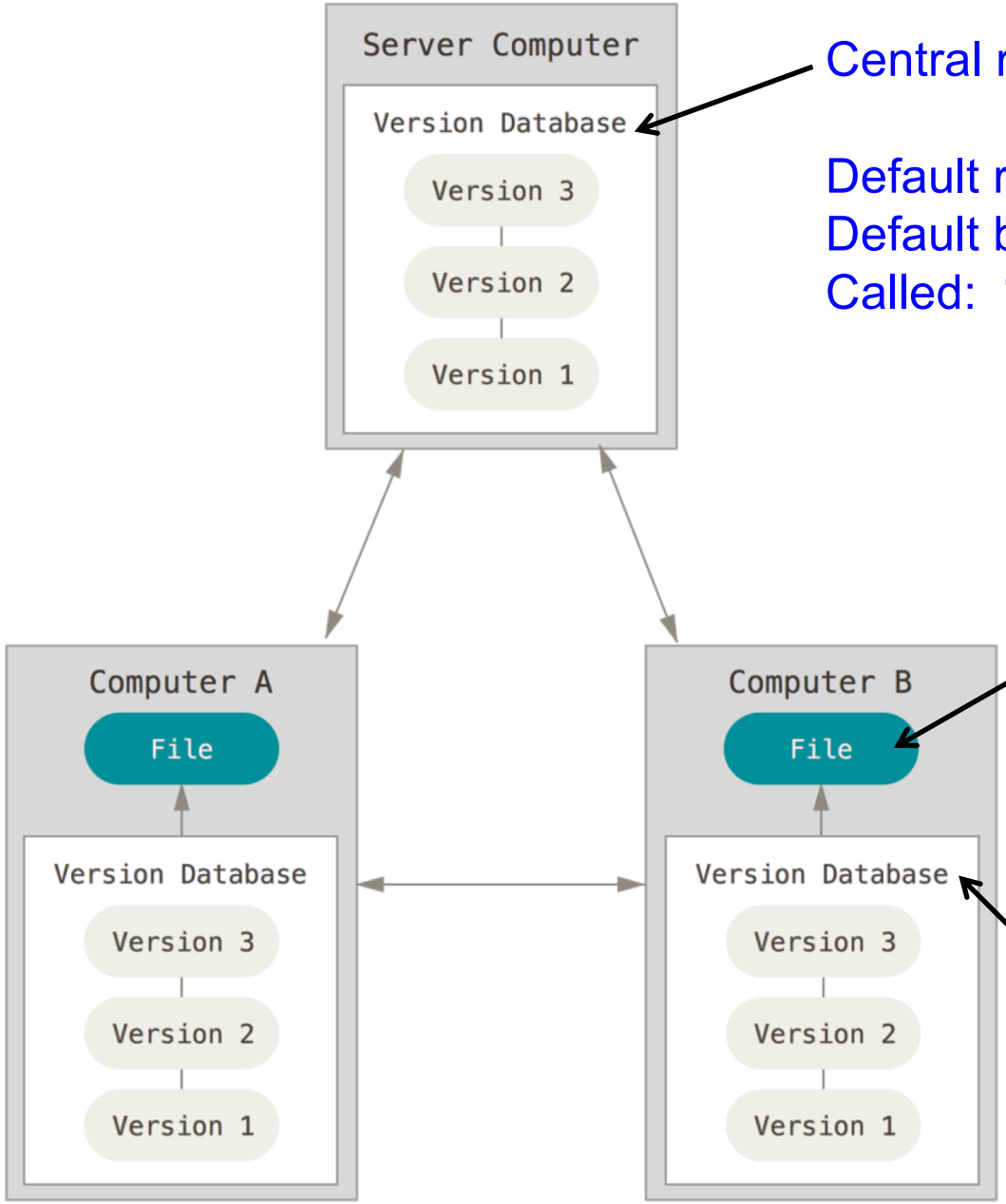
```
[uranus:391test lemieux$ git pull
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 3), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/ubc-guy/391test
   a6b35ca..29340d2  main       -> origin/main
Updating a6b35ca..29340d2
Fast-forward
 file.h   | 1 +
 primes.c | 2 ++
 2 files changed, 3 insertions(+)
uranus:391test lemieux$
```




- Note: "`git pull`" retrieves changes to shared repository and applies them locally
- If I had also modified primes.c, there would be a conflict – need to merge results

# Git – Decentralized Control



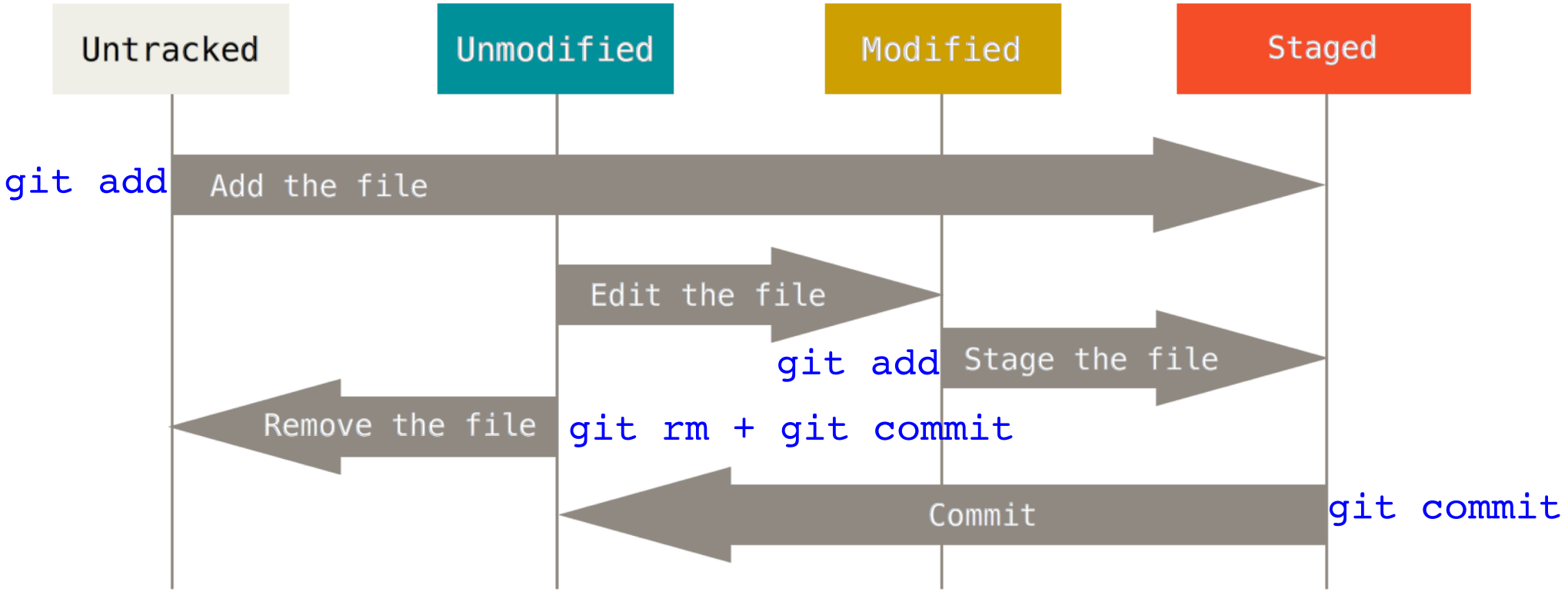

Central repository, eg: github

Default remote name: "origin"
Default branch: "main"
Called: "origin/main"

Your working file set (edit, compile, run)

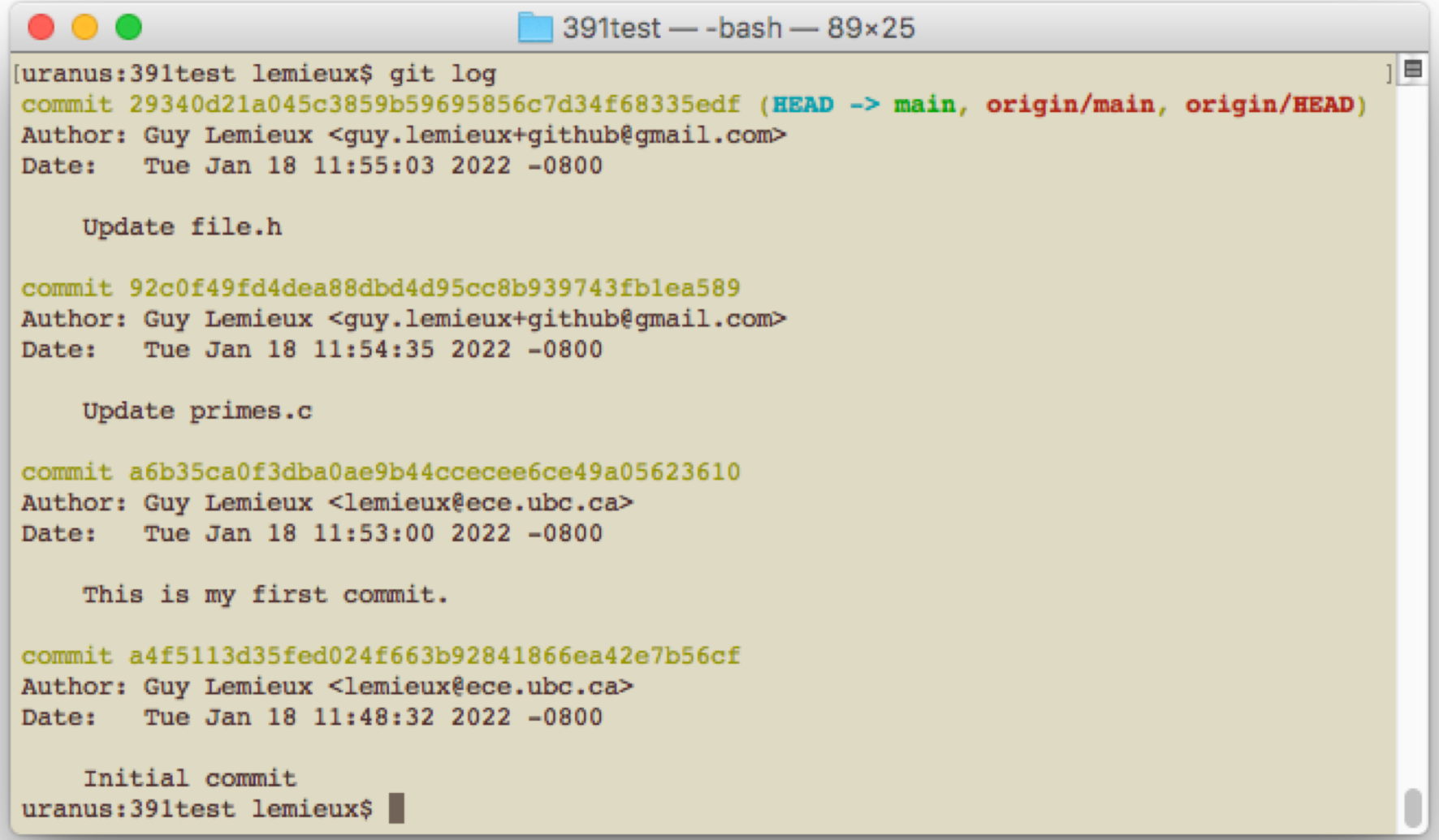
Your personal repository/history

# Git – File Life Cycle




```
Also:  git mv oldname.c newname.c
       git commit
```

# Git – Viewing History



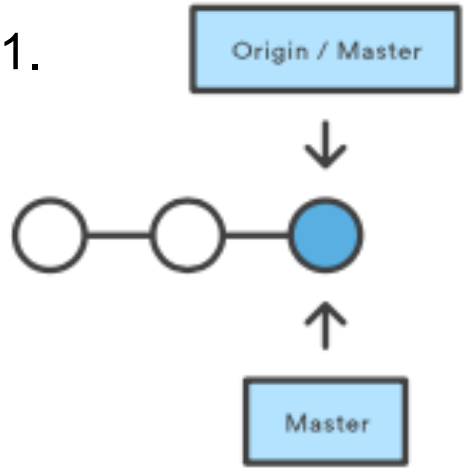
# Git – Summary So Far

```
$ git clone https://github.com/name/project.git
$ cd project
(create/modify files)
$ git add file1.c file2.c file.h
(create/modify primes.c)
$ git add primes.c
$ git commit
$ git push
$ git status

<...somebody else works on files...>

$ git pull
```

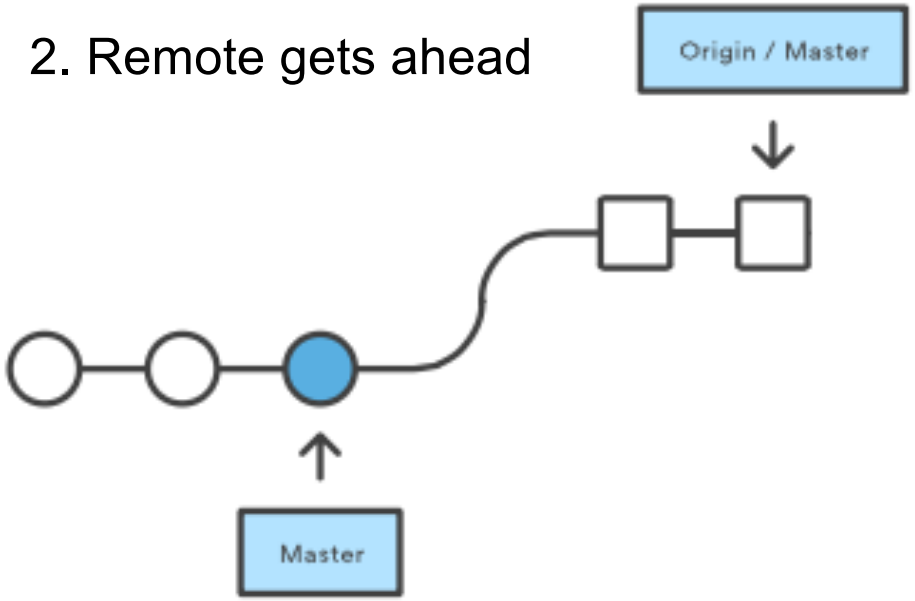
# Git Pull – moving heads

- Git pull

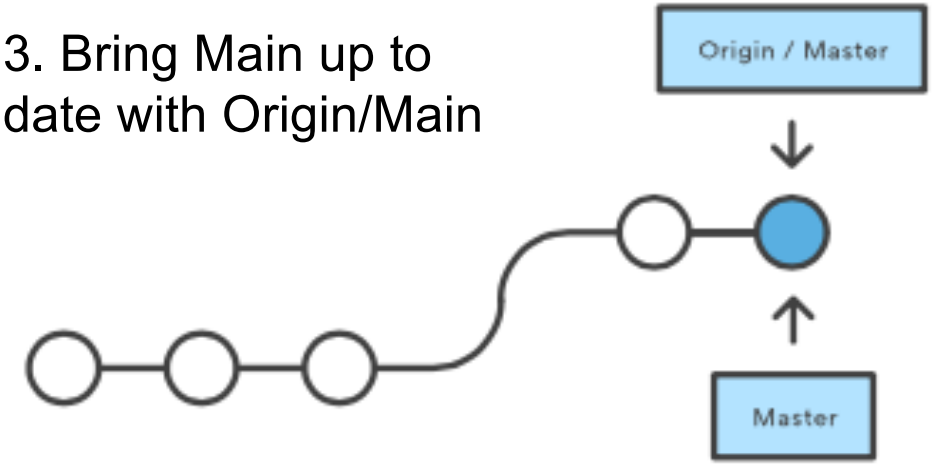
1.




2. Remote gets ahead




3. Bring Main up to date with Origin/Main

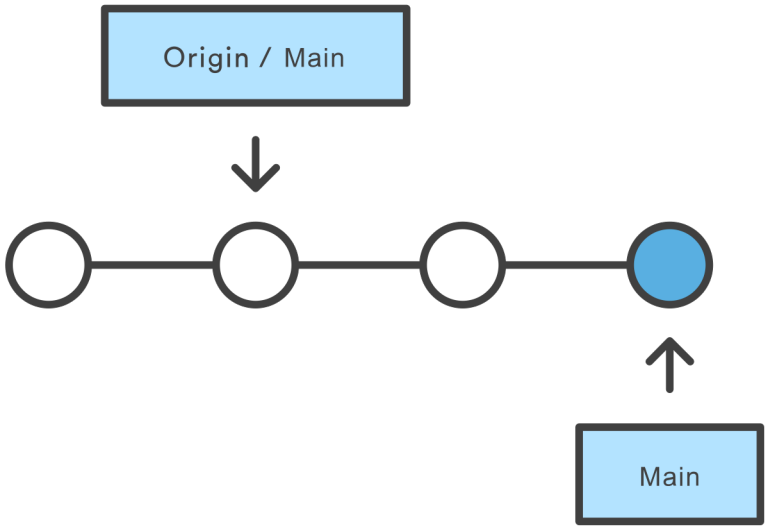



Repositories created before Oct 2020 used “Master”.

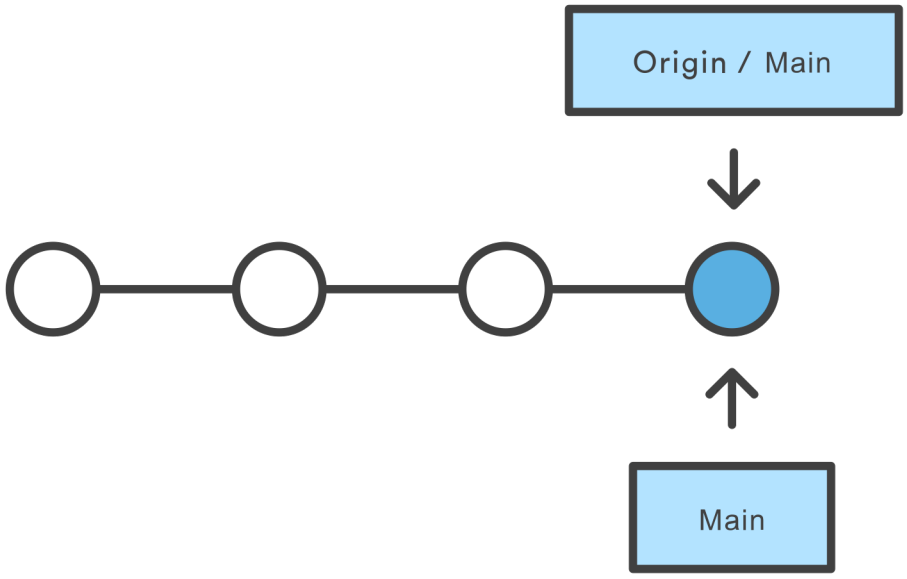
Now, they use “Main”.

# • Git Push – moving heads

- Before pushing




- After pushing



# Git – Throwing away changes

- You started to modify file1.c
  - Decided your approach was wrong
  - Want to start over with the last committed (staged) file1.c
    - Even if you (accidentally) removed file1.c
  - Restore it with the version from local repository

```
$ rm file1.c
$ git checkout file1.c
```

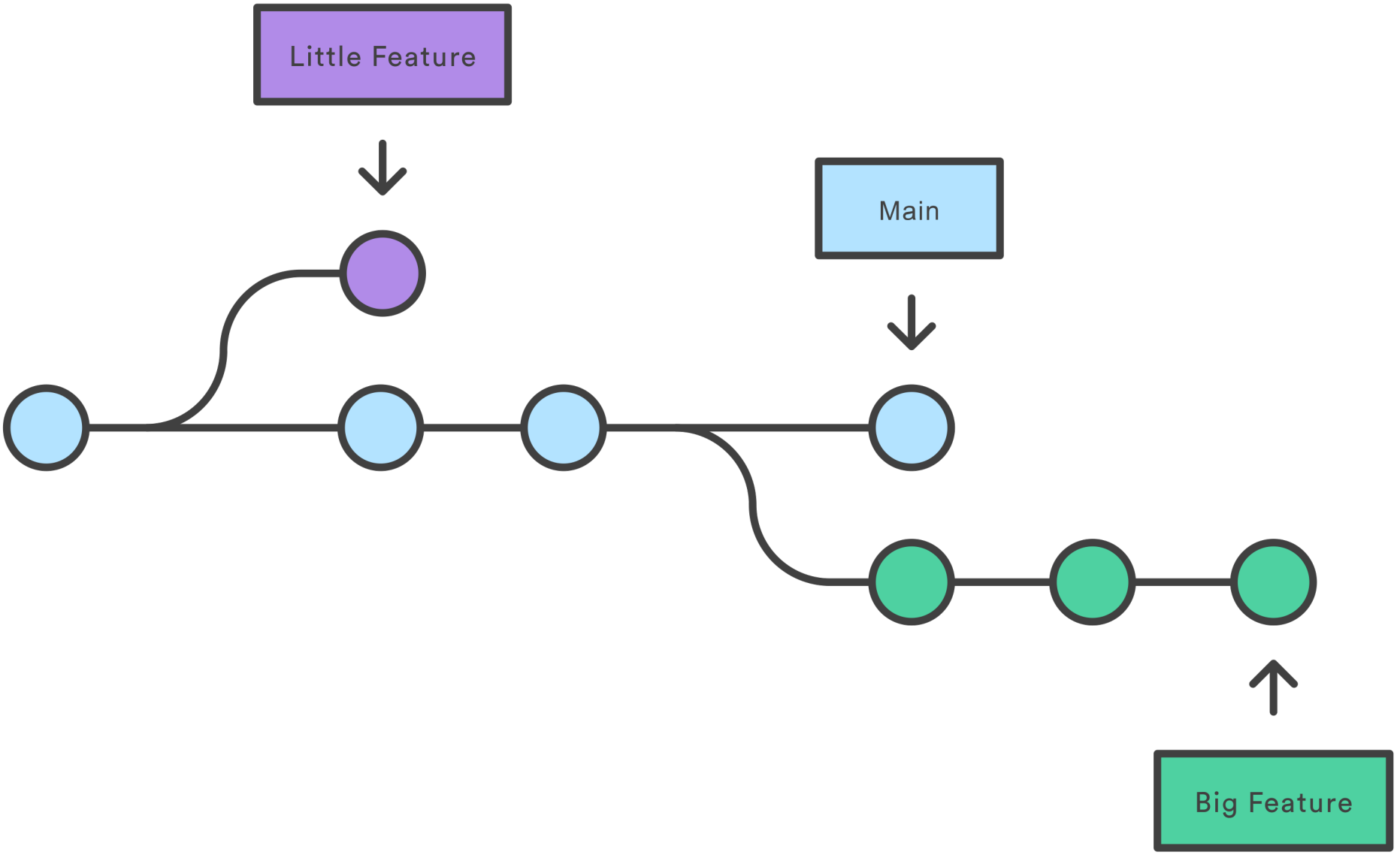
# Git – Branches

- By default, you are working on “Main” branch
  - You should never do this!

- Instead
  - Create new branches
  - Merge them back into Main

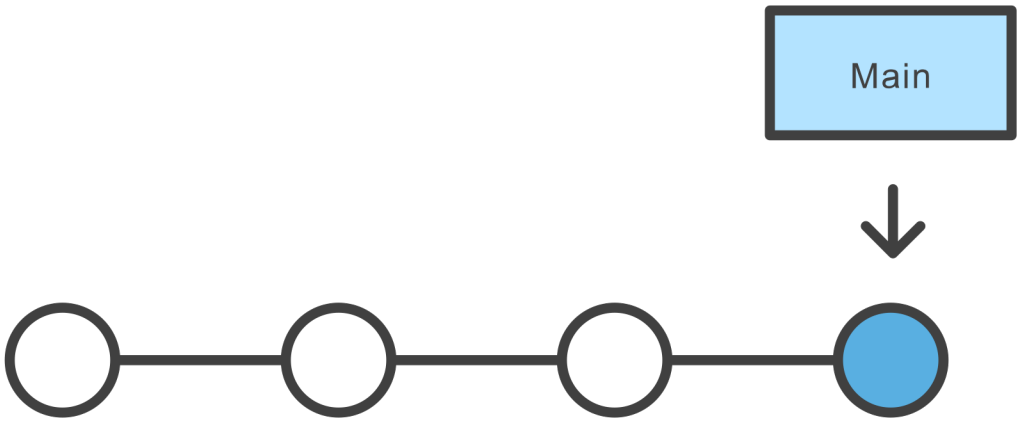
- Need a workflow

# Git – Branches



# Git – Creating Branches

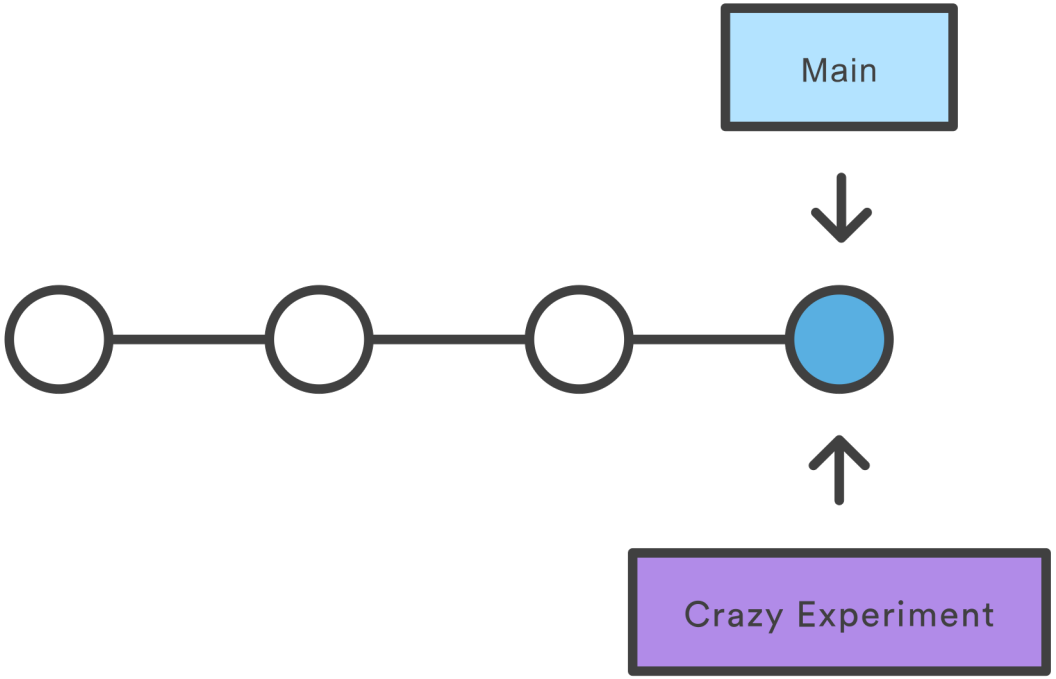
Before:




```
$ git branch crazy-experiment
```

(creates branch in repo)

After:



# Git – Checking Out a Branch

```
$ git branch some-feature
```
(creates branch in repo)

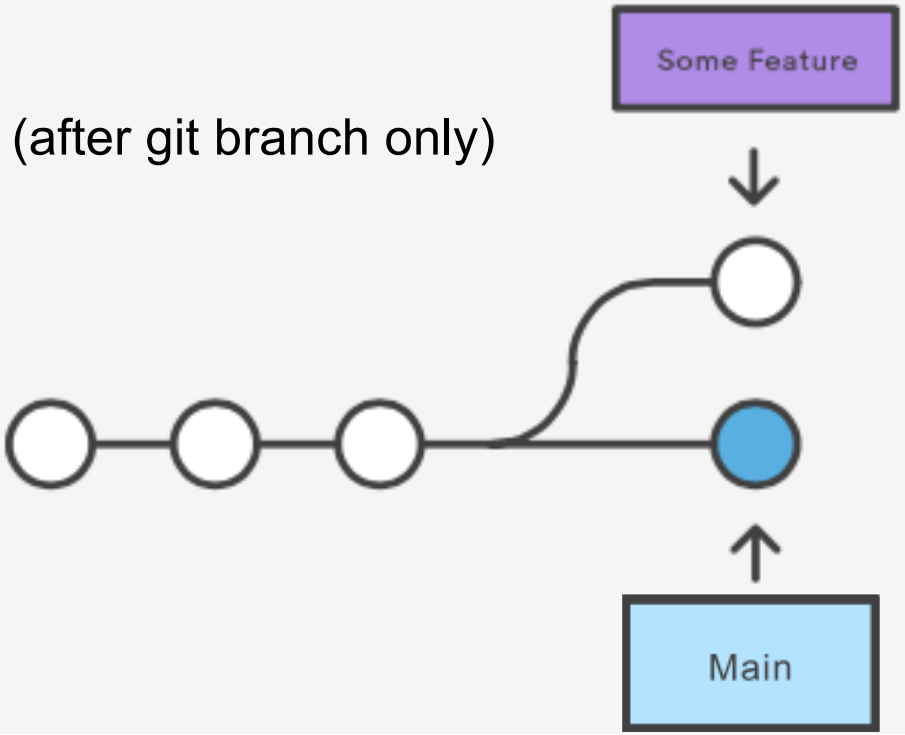
```
$ git checkout some-feature
```
(selects branch for commits)

(or)

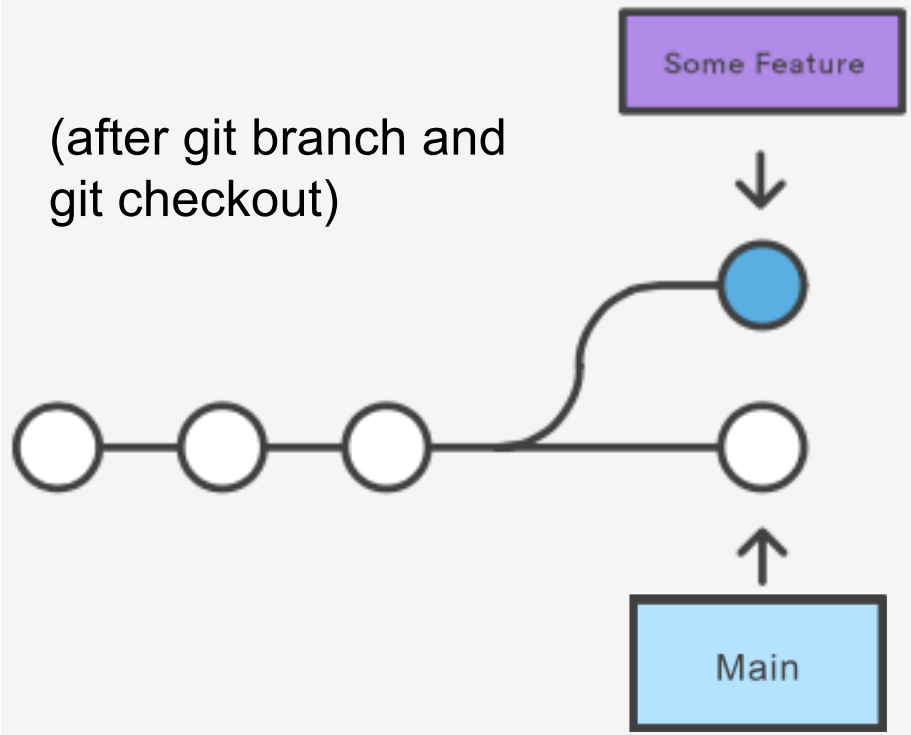
```
$ git checkout –b some-feature
```
(does both in one cmd)

```
To send this new local branch to the remote repository:
$ git push –u origin some-feature
```

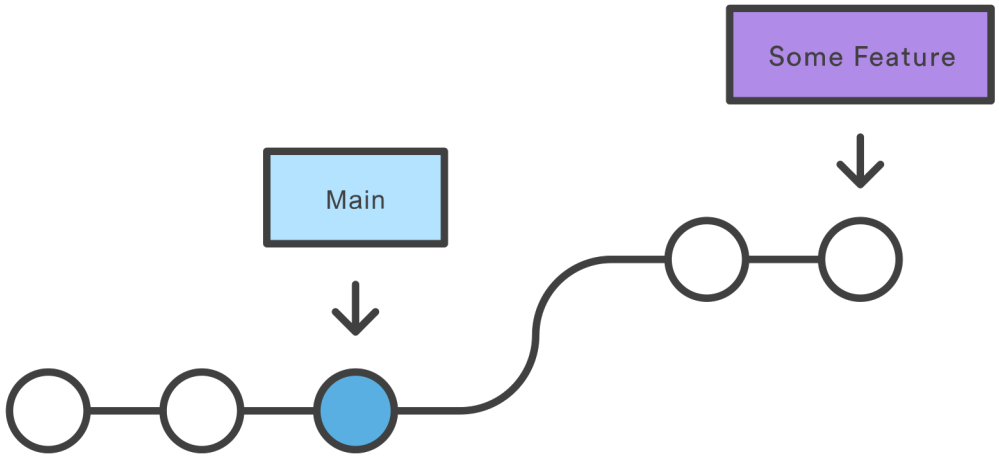
(after git branch only)




(after git branch and git checkout)



# Git – Merge

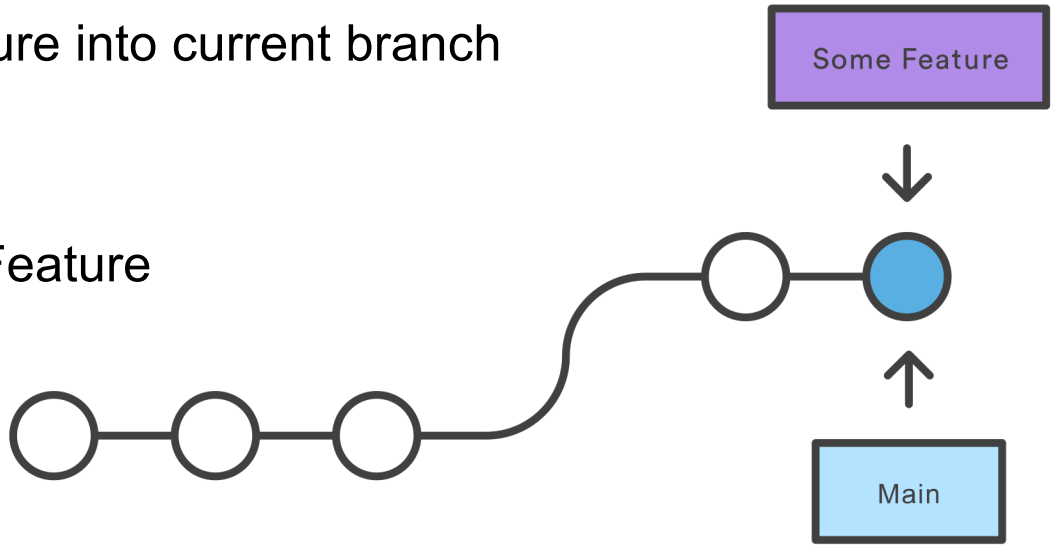



```
$ git checkout main          (select Main branch for commits)
$ git merge some-feature     (fast-forward Main)
```

Merge brings contents of some-feature into current branch

Simple case:

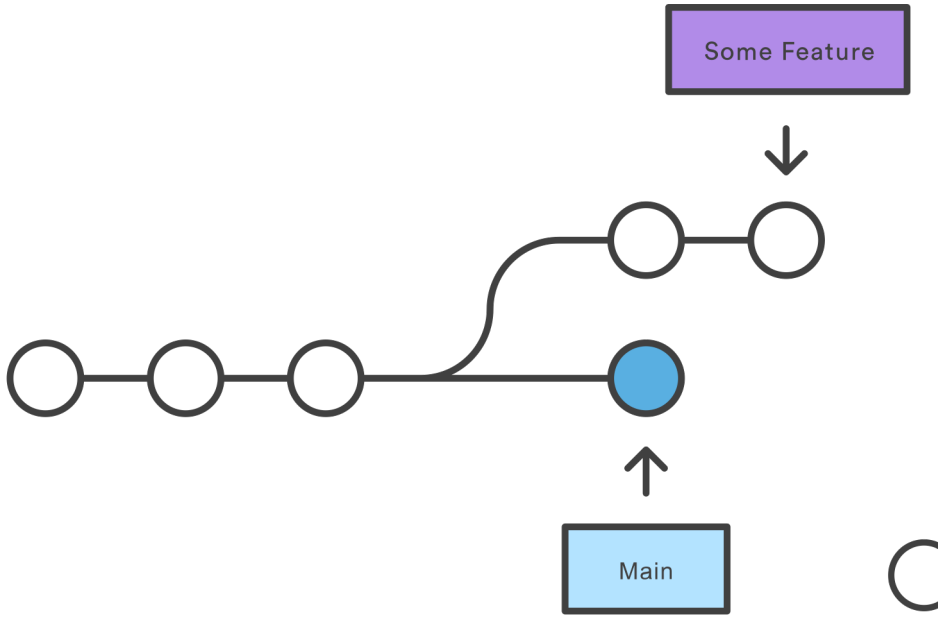
- Linear path from Main to Some Feature
- Called a fast-forward merge

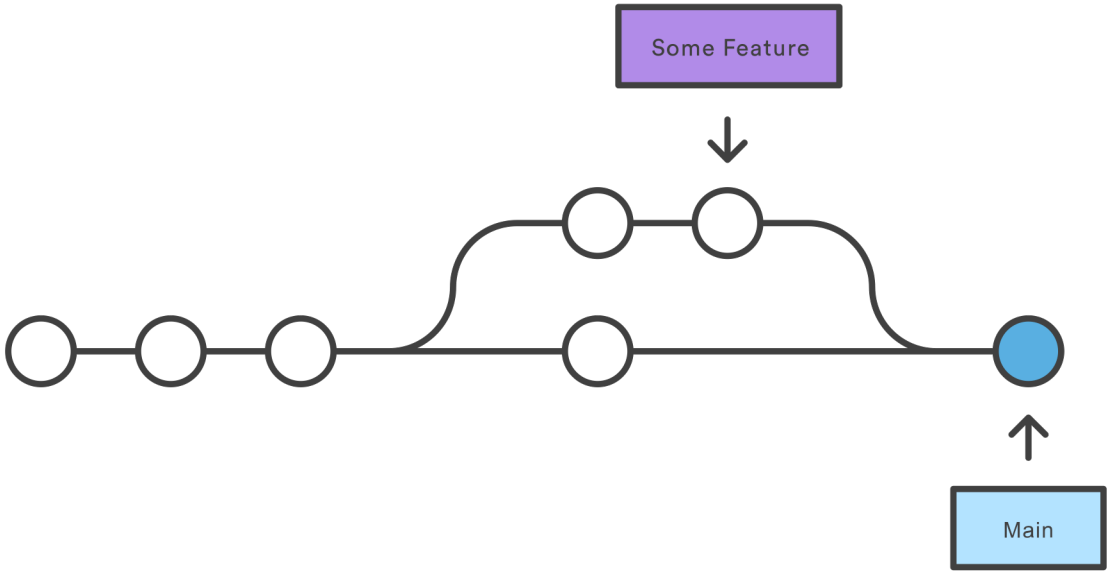


# Git – Merge

```
$ git checkout main   (select Main branch for commits)
$ git merge some-feature
```

- No linear path from Main to some-feature, called 3-way merge
  - Git uses 3 commits to complete (Main, some-feature, then merged Main)
  - Conflicts may arise



# Git – 3-way Merge

- General process

1. Merge Main into Some Feature branch

```
$ git checkout some-feature
$ git merge main
```

2. Test branch, fix errors, commit branch

```
(run, edit, modify)
$ git add changed1.c changed2.c
$ git commit
```

Do steps 1,2 first

Allows you to debug branch with latest version of Main safely, without breaking Main.

3. Merge Some Feature branch into Main

```
$ git checkout main
$ git merge some-feature
```

4. Test Main, fix errors, commit Main

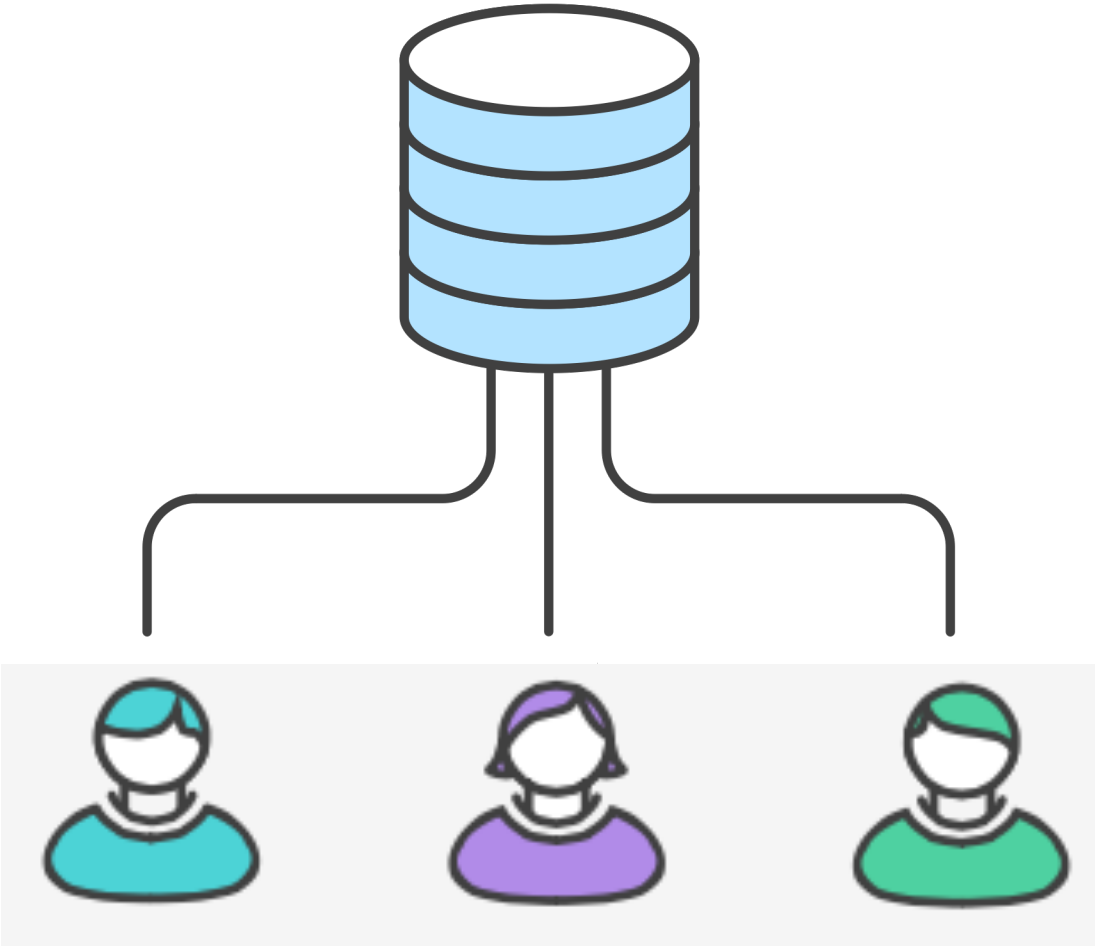
```
(run, edit, modify)
$ git add changed3.c changed4.c
$ git commit
```

Do steps 3,4 second

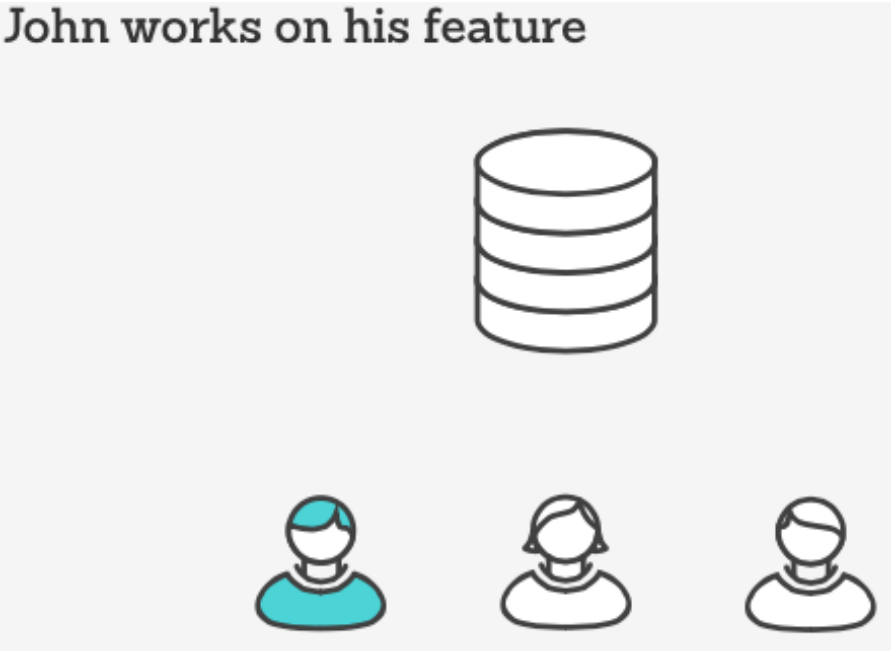
Since branch was just tested, and there were no further changes to Main, there should be no problems here.

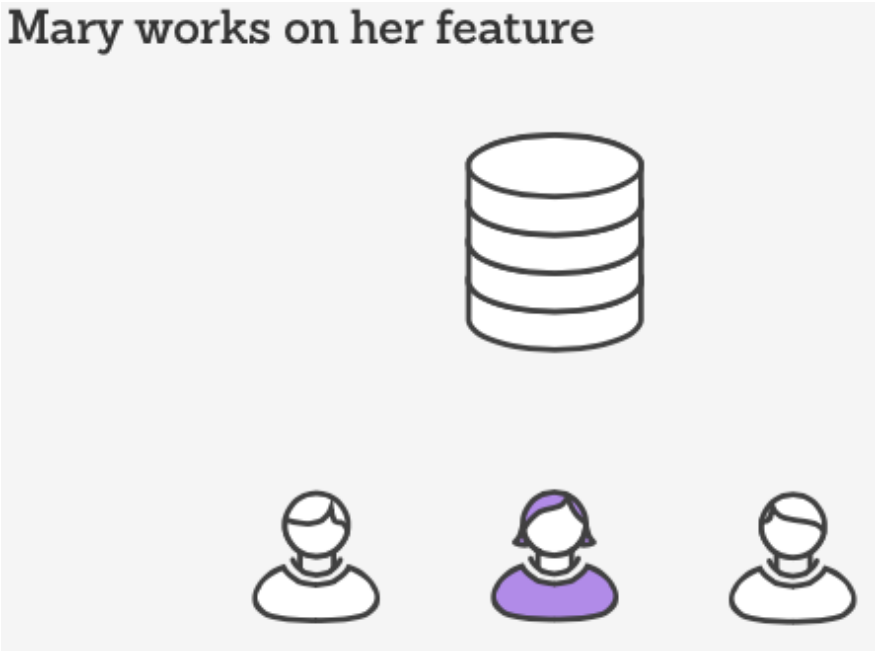
# Git – Workflow

Everybody clones the central repository

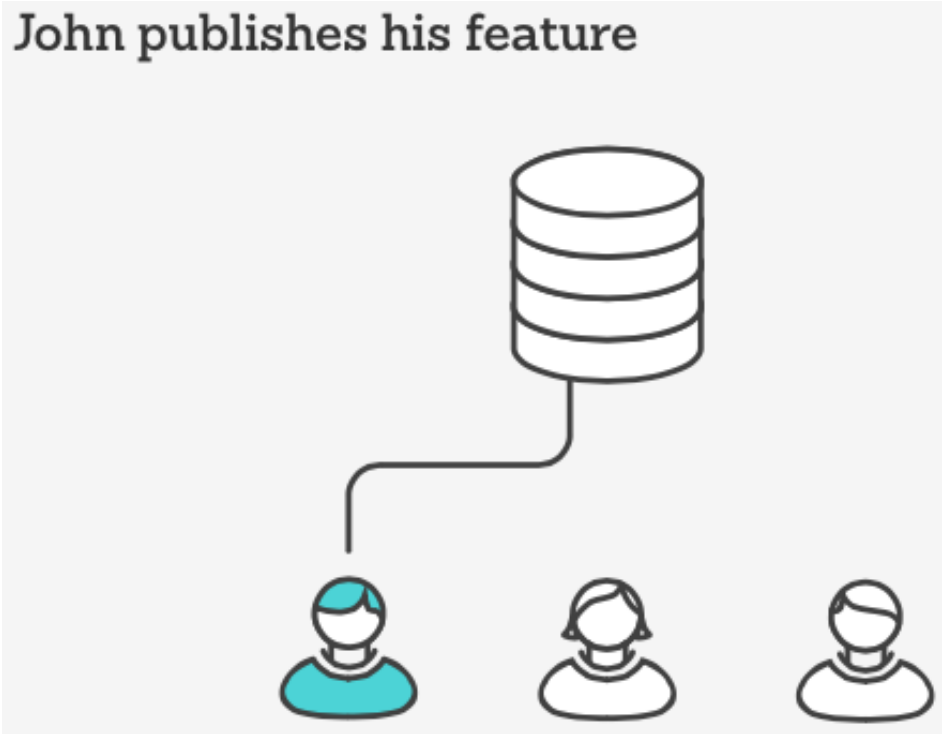


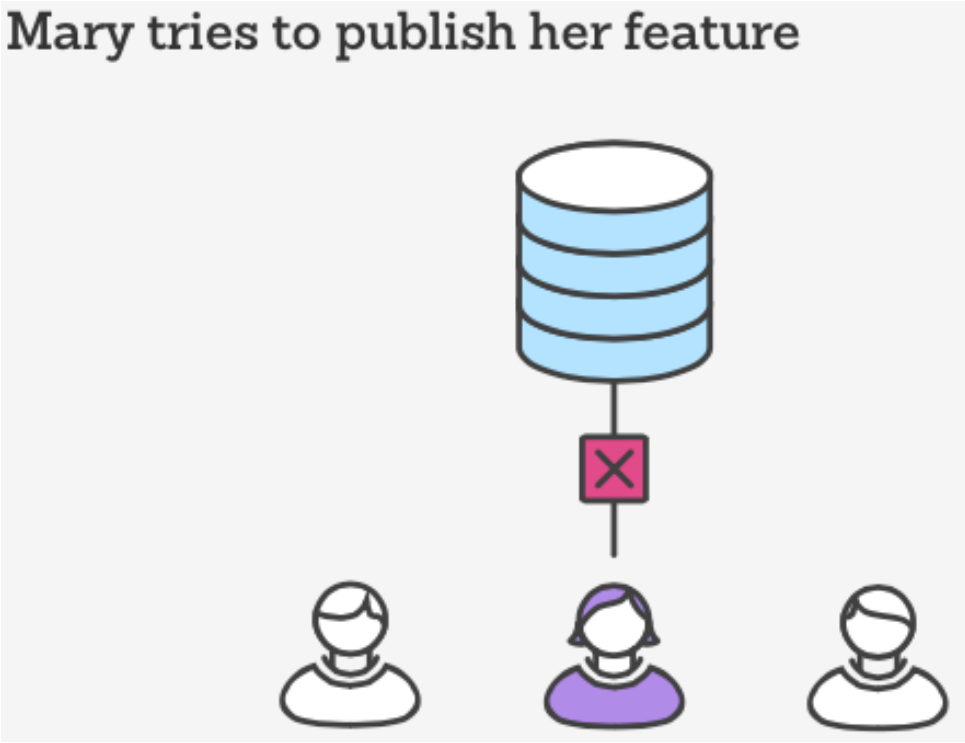
Working in parallel…



```
(john) $ git push origin main
(mary) $ git push origin main
```

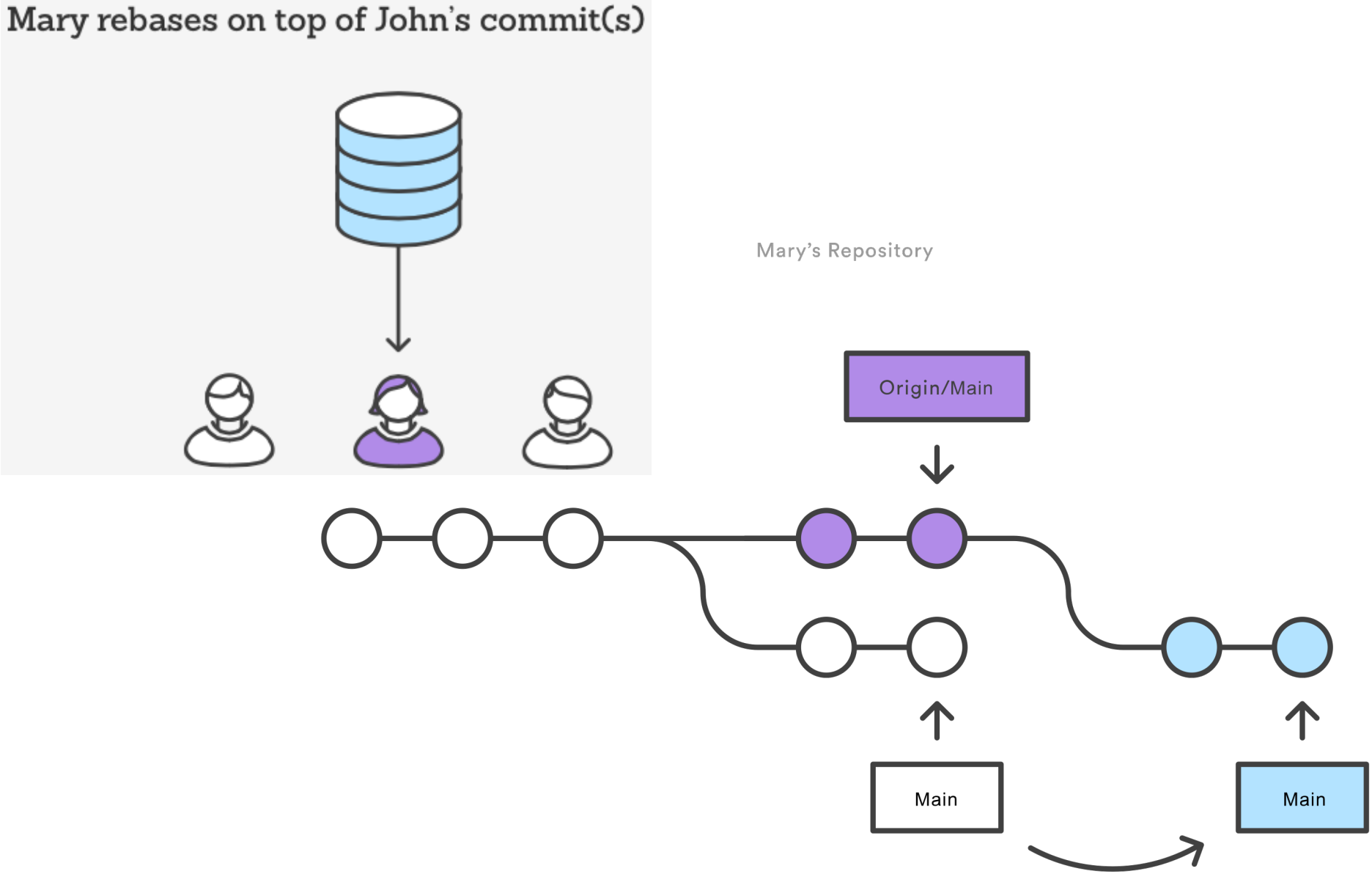



```
error: failed to push some refs to '/path/to/repo.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

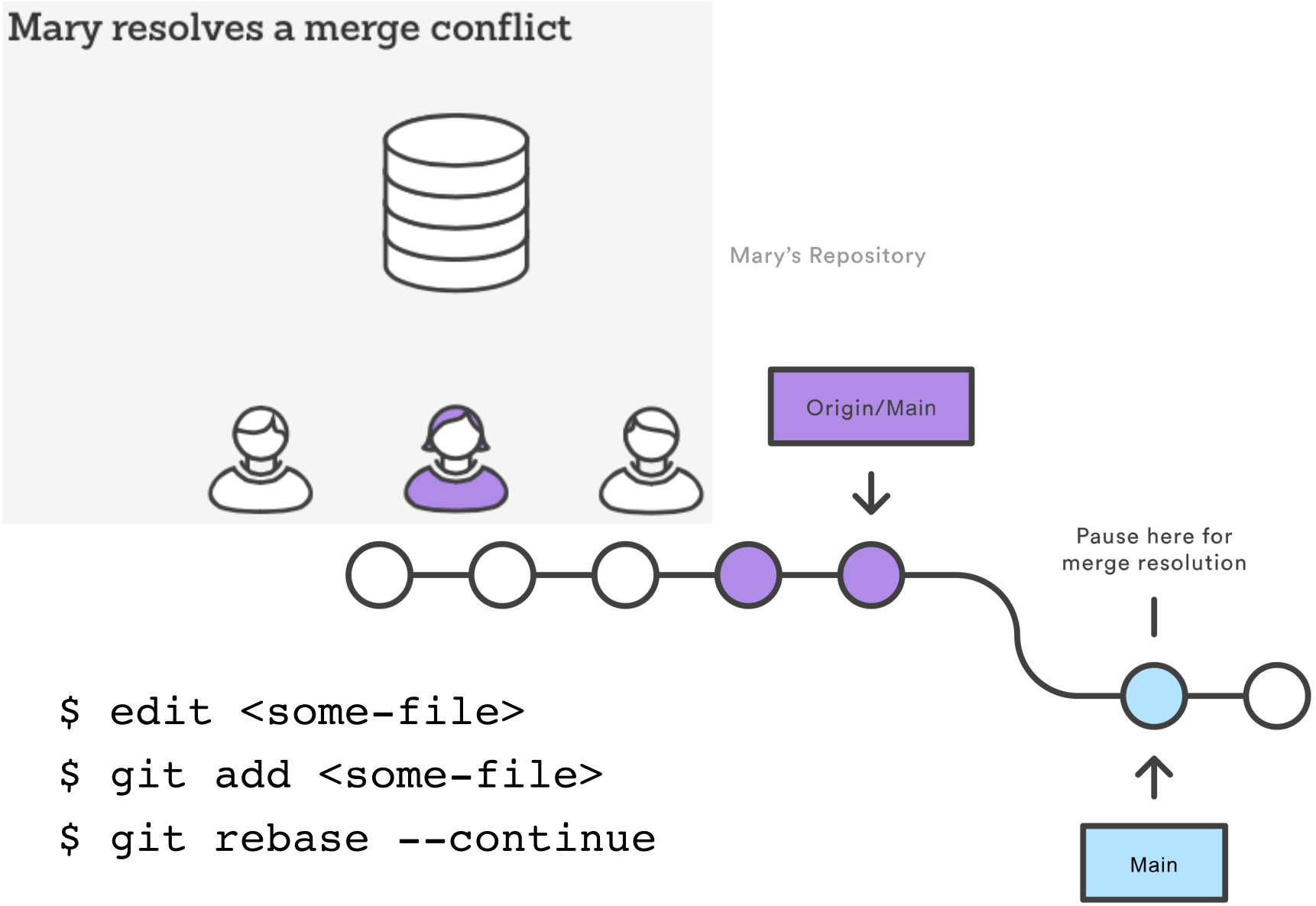
# Mary moves her work “after” new Origin/Main

```
(mary) $ git pull --rebase origin main
```



# Merge Conflicts – Manual Resolution

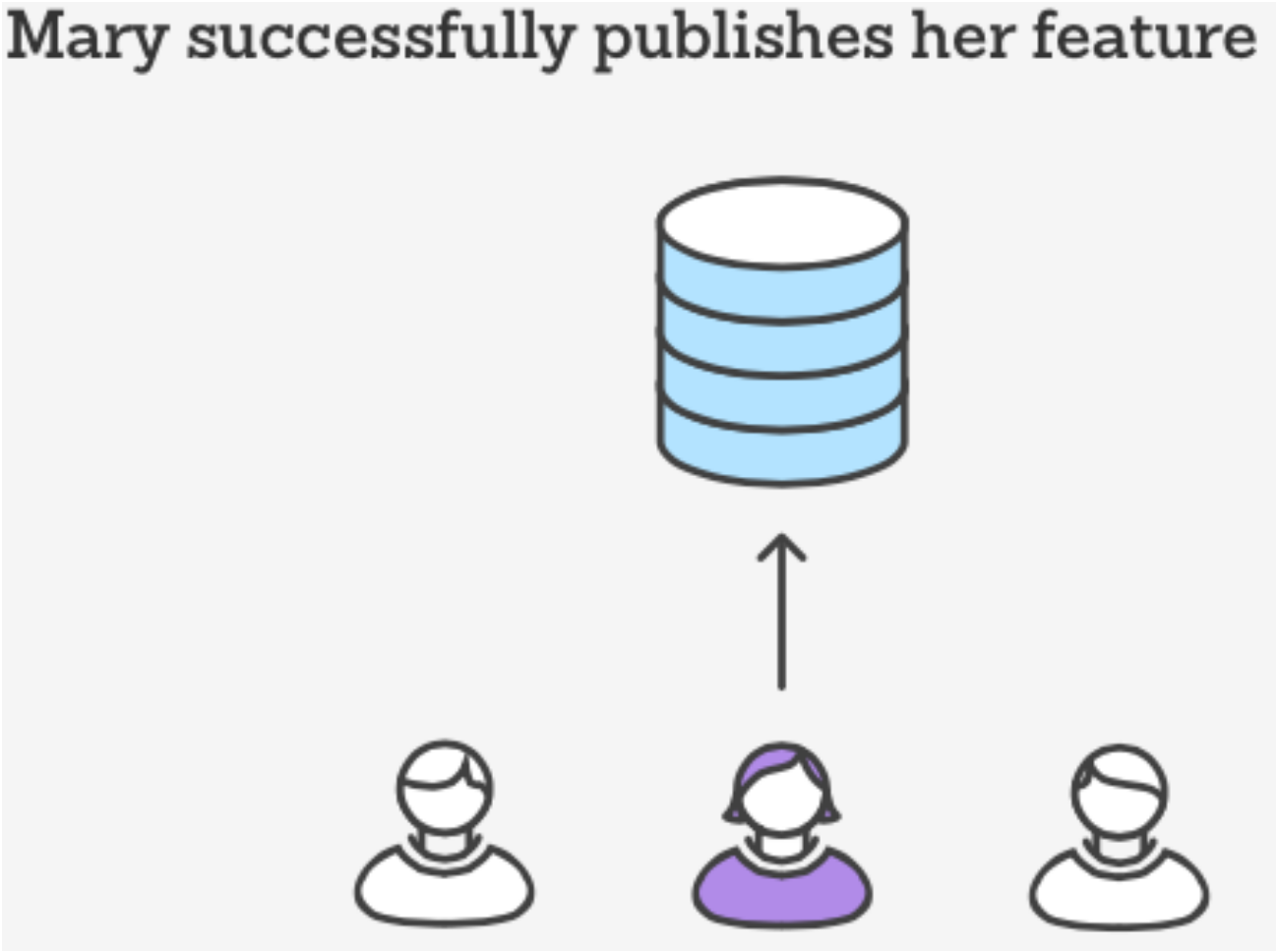
```
CONFLICT (content): Merge conflict in <some-file>
```




```
$ edit <some-file>
$ git add <some-file>
$ git rebase --continue
```

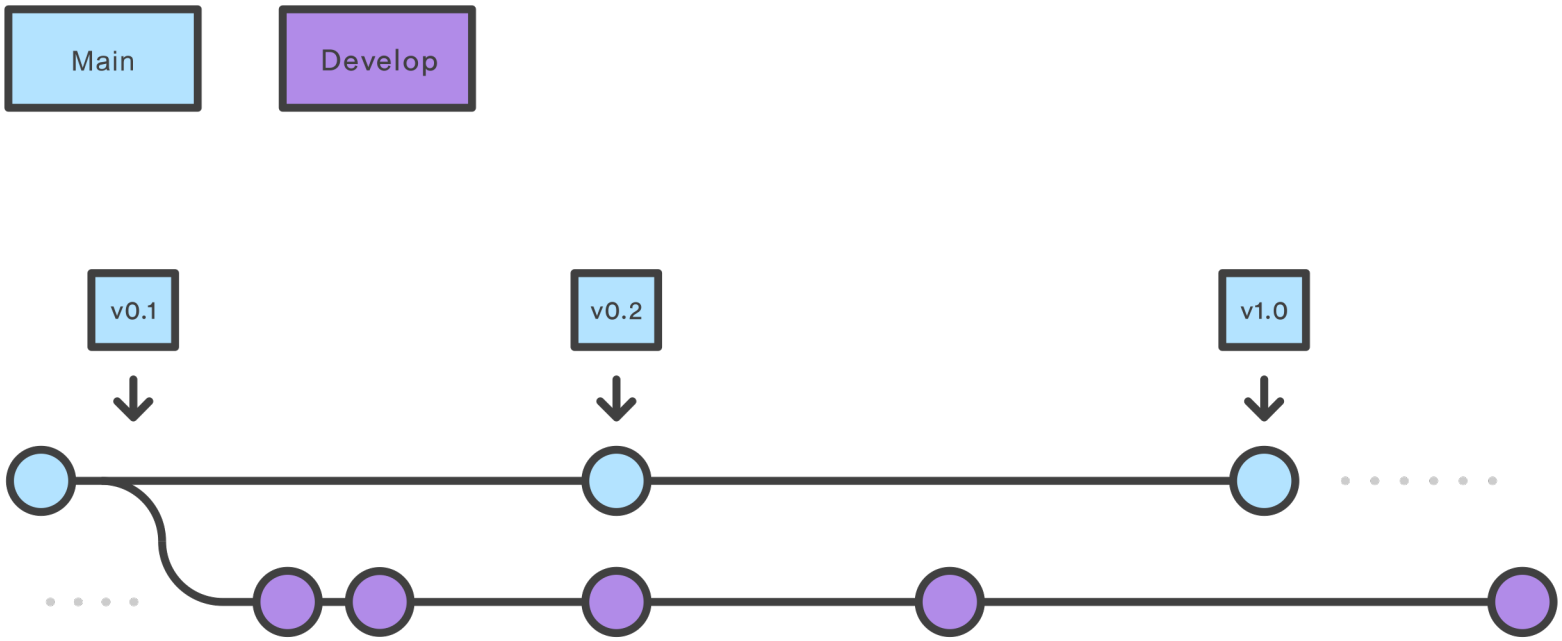
# After Mary Has Successfully Integrated John's Push

```
(mary) $ git push origin main
```



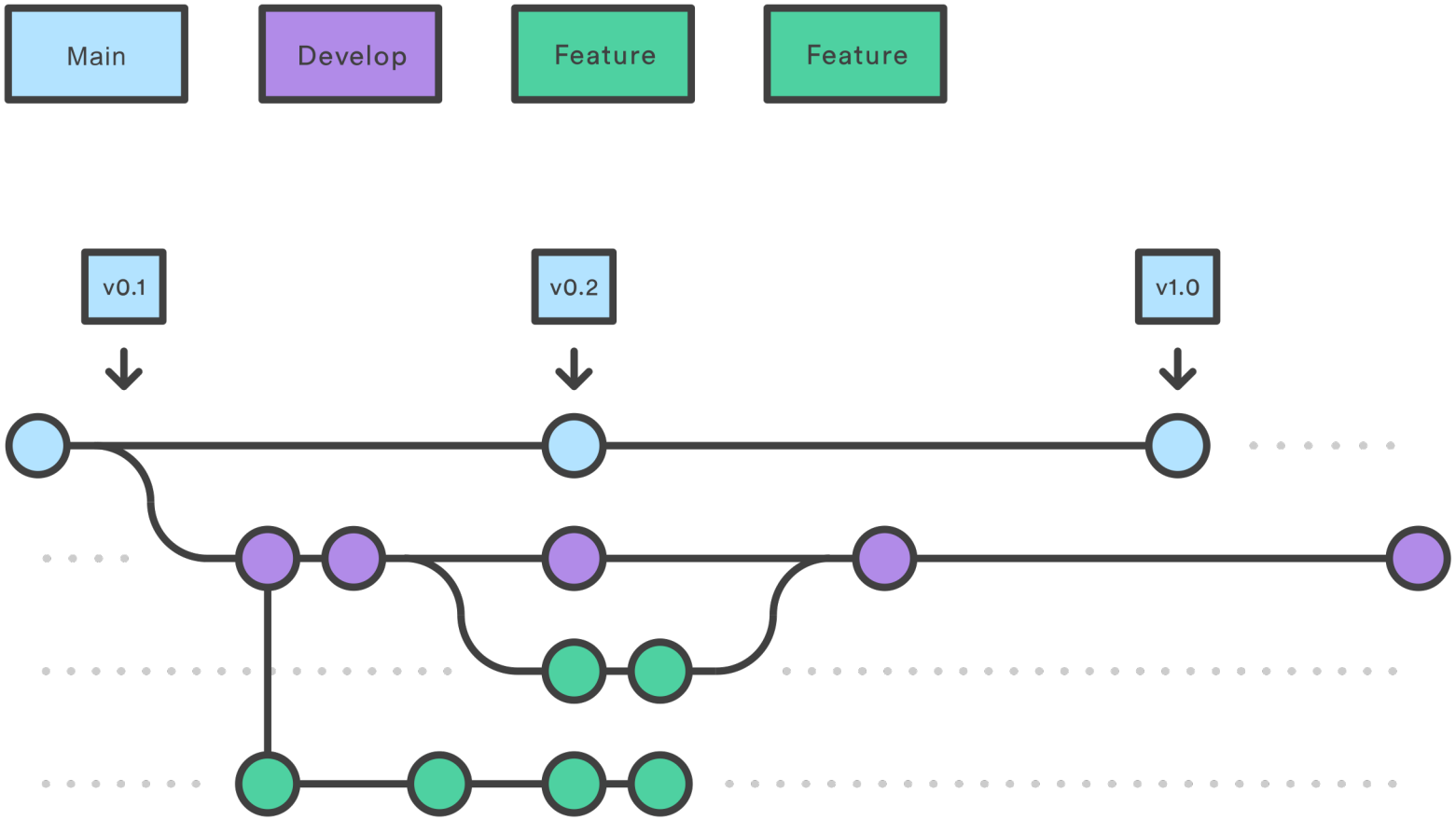
# Git –Workflow

- Never work directly in Main branch
  - This is only used to hold released versions
  - Any version in Main ALWAYS works (official release, no bugs)
- Immediately create a “develop” branch
  - Create feature branches off “develop”
  - Merge completed features back in to “develop”



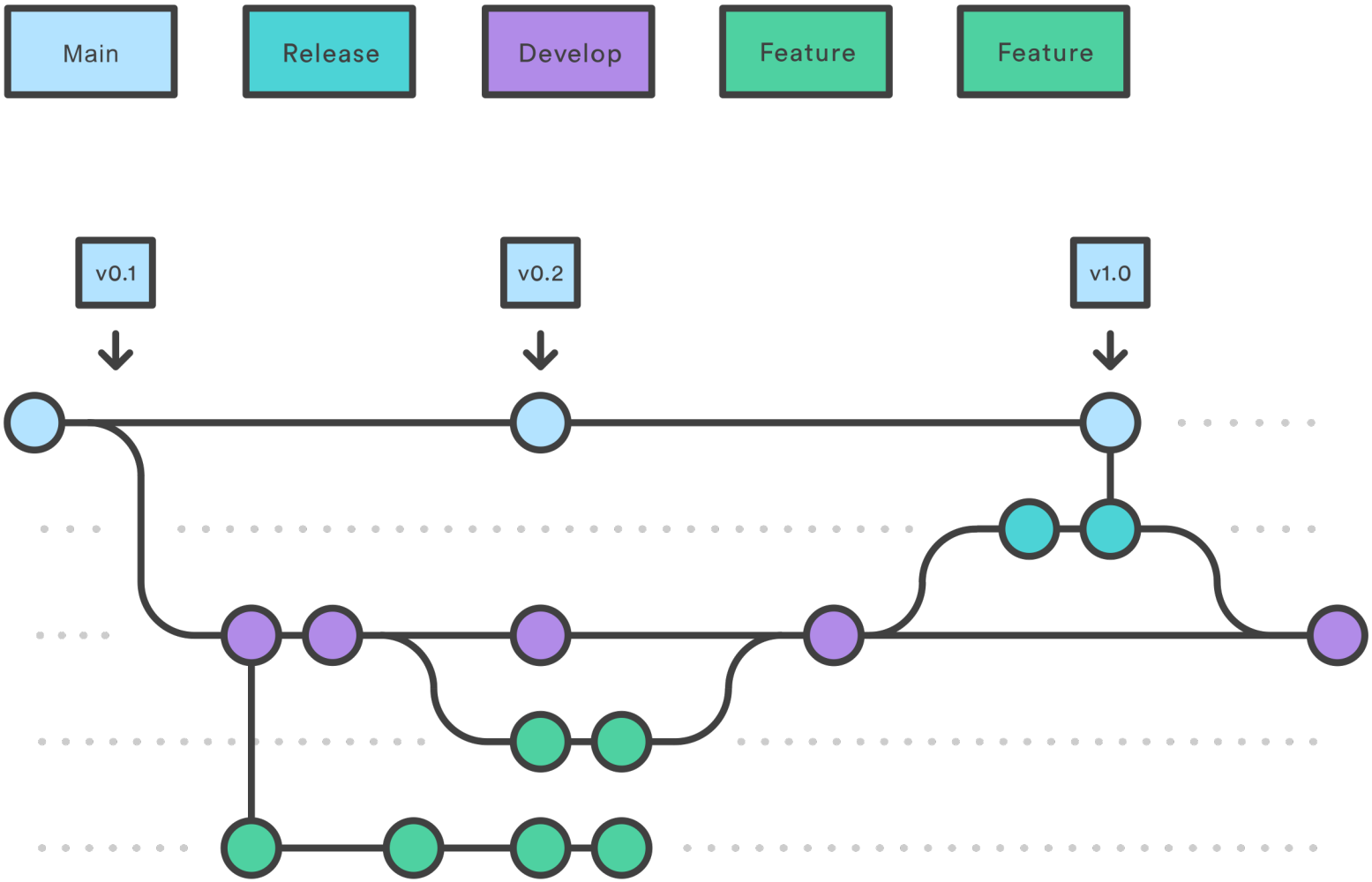
# Features as Branches

- Feature branches always fork off of “develop”
  - And merge them back into “develop” as well



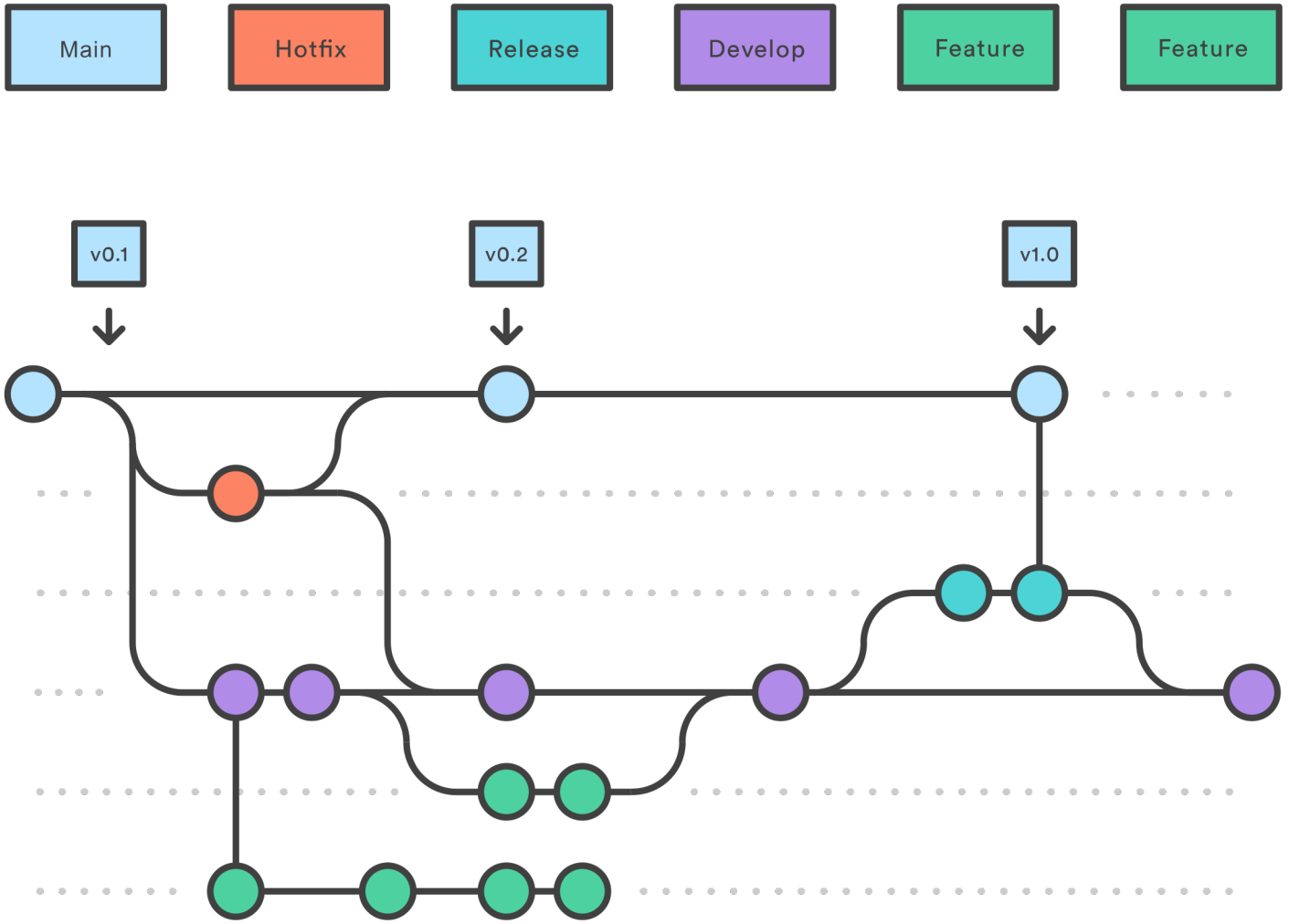
# Release Candidates as a Branch

- Create “Release Candidate” branches
  - Merge these into Main when ready



# Quick Bugfixes as a Branch

- Maintenance branches, or “hotfixes”
  - Branch off Main
  - Merge into both Main and “develop”



# Git – Best Practices

1. **Update/pull every morning**
   - Keeps your local copy inline with revisions done by your team
   - Forces you to test your version against the latest version
   - For some bugs, must not pull until it's fixed!
2. **Commit/push early and often**
   - Do not code for 2 weeks then expect to push
   - Others will have heavily modified repository, your code will be out of date and require many changes
3. **Commit logical units**
   - Each commit should hold the minimal set of files / changes to fix one issue / add one feature
   - Use separate commits for different issues/features
   - Do not commit individual files, only complete filesets
4. **Don't break the build (unit tests must run)**
   - Always run tests before committing!!!

# Git – Hardware and Software

Possible directory structure, all of it in Git!

Note: do not store tool-generated output files in repository

- project
  - docs
  - hardware
    - de1system
    - verilog
      - component1
      - component2
    - testbenches
  - scripts
  - software
    - lib
    - test
    - hwtest
    - src
      - software1
      - software2

# Git – More info ?

- Great GIT tutorials
  - http://www.atlassian.com/git/tutorials
  - Eg: “Using Branches” and “Comparing Workflows”

- Official documentation
  - http://git-scm.com/doc

# Pull Requests

- Pushing your changes to origin/main or origin/develop can be destructive
  - Forces others to adopt your changes
  - Your change may break things, prevents others from pushing

- Alternative: **pull request**
  - Email your team, ask them to pull your changes when they are ready
    - This is not actually a git command
    - Also used to ask for feedback

  - Example steps:
    - Push your feature-branch: `git push –u some-feature`
    - Email team with git hash + branch name (some-feature)
    - Feedback requested: member(s) comment on it
    - Merge requested:
      - One member merges/tests/commits into the common shared branch (such as origin/develop)
      - Everyone pulls the committed change

# Newer git commands

New git commands to investigate:

- `git worktree`
  - Manage multiple working trees
  - Checks out more than one branch
- `git restore`
  - Restores working tree files
- `git switch`
  - Switches branches