Git

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand why we use version control
- Understand the basic git workflow and GitHub
- To use the basic git commands
- Understand the advanced git workflow

Inspiration

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning. – Rich Cook

Resources

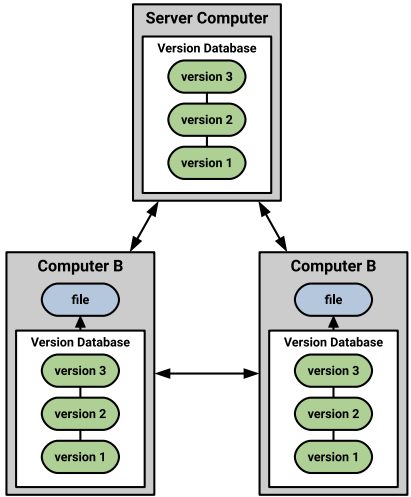
- Vincent Driessen's A Successful Git Branching Model
- Atlassian's Gitflow Workflow
- Semantic Versioning

Why Version Control

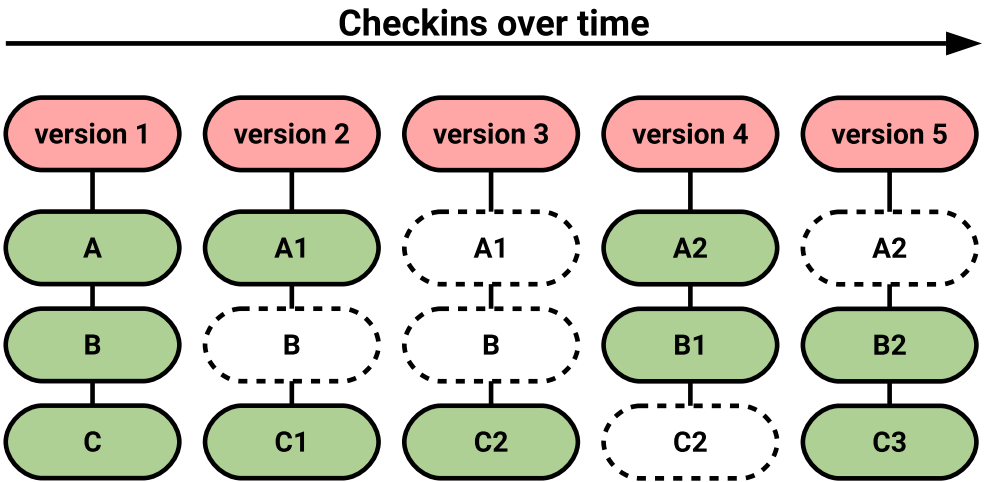
Git History

- Came out of Linux development community
- Linus Torvalds, 2005
- Initial goals:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like Linux efficiently

Git Uses a Distributed Model



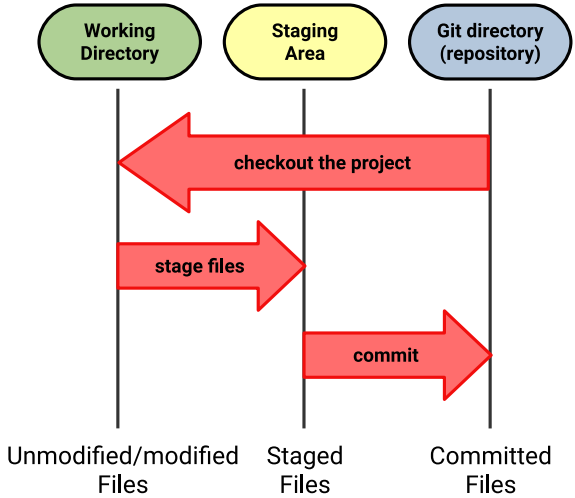
Git Takes Snapshots



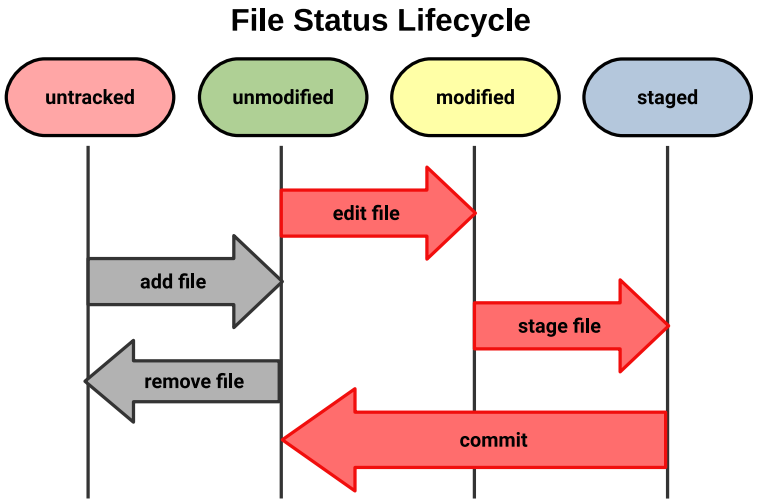
Git Uses Checksums

- Git generates a unique SHA-1 hash for every commit
  - 40 character string of hex digits
- Refer to commits by this ID rather than a version number
- Often we only see the first 7 characters:
  - `1677b2d Edited first line of readme`
  - `258efa7 Added line to readme`
  - `0e52da7 Initial commit`

Local Projects



Git File Lifecycle



Basic Workflow

Basic Git workflow:

1. **Modify** files in your working directory.
2. **Stage** files, adding snapshots of them to your staging area.
3. Do a **commit**, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

- Notes:
  - If a particular version of a file is in the **git directory**, it's considered **committed**.
  - If it's modified but has been added to the **staging area**, it is **staged**.
  - If it was **changed** since it was checked out but has not been staged, it is **modified**

What is GitHub?

- GitHub.com is a site for online storage of Git repositories.
- Many open source projects use it, such as the Linux Kernel.
- You can get free space for open source projects or you can pay for private projects.

Question: Do I have to use github to use Git? **Answer**: No!

- you can use Git completely locally for your own purposes, or
- you or someone else could set up a server to share files, or
- you could share a repo with users on the same file system.

Using Git

Get Ready to Use Git!

❶ Set the name and email for Git to use when you commit:

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

- You can call `git config -list` to verify these are set.
- These will be set globally for all Git projects you work with.
- You can also set variables on a project-only basis by not using the `--global` flag.
- You can also set the editor that is used for writing commit messages:

`$ git config --global core.editor emacs` (it is vim by default)

Create a Local Copy

❷ Two common scenarios: (only do one of these)

- To **clone an already existing repo** to your curent directory: `$ git clone <url> [local dir name]`
  This will create a directory named local dir name, containing a working copy of the files from the repo, and a **.git** directory (used to hold the staging area and your actual repo).
- To **create a Git repo** in your current directory: `$ git init`
  This will create a **.git** directory in your current directory. Then you can commit files in that directory into the repo:

```
$ git add file1.Java
$ git commit -m "initial project vesion"
```

Git Commands

command	description
`git clone` *`url [dir]`*	copy a git repository so you can add to it
`git add` *`files`*	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` *`[command]`*	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others	`init`, `reset`, `branch`, `checkout`, `merge`, `log`, `tag`

Committing Files

- The first time we ask a file to be tracked, and **every time before we commit a file** we must add it to the staging area:

```
$ git add README.txt hello.java
```

This takes a snapshot of these files at this point in time and adds it to the staging area.

- To move staged changes into the repo we commit:

```
$ git commit -m "Fixing bug #22"
```

```
Note: To unstage a change on a file before you have committed it:
`$ git reset HEAD -- filename`

Note: To unmodify a modified file:
`$ git checkout -- filename`
```

Note: These commands are just acting on **your local version of repo**

Status and Diff

- To view the **status** of your files in the working directory and staging area:

`$ git status` or `$ git status -s` (-s shows a short one line version)

- To see what is modified but unstaged:

```
$ git diff
```

- To see staged changes:

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

Viewing Logs

To see a log of all changes in your local repo:

- `$ git log`
- `$ git log --oneline` (to show a shorter version)

```
1677b2d Edited first line of readme
258efa7 Added line to readme
0e52da7 Initial commit
```

- `$ git log -5` (to show only the 5 most recent updates, etc.)

Note: changes will be listed by commitID #, (SHA-1 hash)

Note: changes made to the remote repo before the last time you cloned/pulled from it will also be included here

Pulling and Pushing

Good Practice:

1. **Add** and **Commit** your changes to your local repo
2. **Pull** from remote repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
3. **Push** your changes to the remote repo

To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory

```
$ git pull origin master
```

To push your changes from your local repo to the remote repo:

```
$ git push origin master
```

Notes: `origin` = an alias for the URL you cloned from `master` = the remote branch you are pulling from/pushing to, (the local branch you are pulling to/pushing from is your current branch)

Branching

To create a branch called experimental:

- `$ git branch experimental`

To list all branches: (* shows which one you are currently on)

- `$ git branch`

To switch to the experimental branch:

- `$ git checkout experimental`

Later on, changes between the two branches differ, to merge changes from experimental into the master:

- `$ git checkout master`
- `$ git merge experimental`

Note: `git log --graph` can be useful for showing branches.

Note: These branches are in your local repo!

Idaho State University
Computer Science

Do This:

1. `$ git config --global user.name "Your Name"`
2. `$ git config --global user.email youremail@whatever.com`
3. `$ git clone https://github.com/grifisaa/gitflowtest`

Then try:

1. `$ git log`, `$ git log --oneline`
2. Create a file named userID.txt (e.g., grifisaa.txt)
3. `$ git status`, `$ git status -s`
4. Add the file: `$ git add userID.txt`
5. `$ git status`, `$ git status -s`
6. Commit the file to your local repo: `$ git commit -m "added userID.txt file"`
7. `$ git status`, `$ git status -s`, `$ git log --oneline`

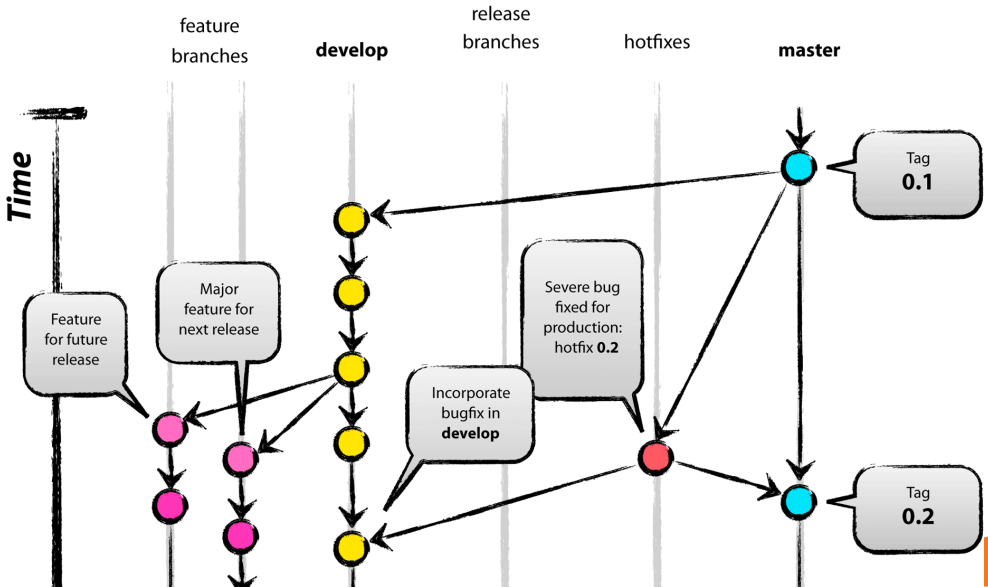
WAIT, DO NOT GO ON TO THE NEXT STEPS UNTIL YOU ARE TOLD TO!!

1. Pull from remote repo: `$ git pull origin master`
2. Push to remote repo: `$ git push origin master`

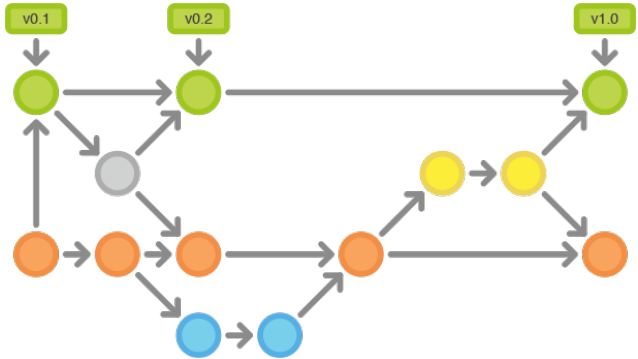
ROAR

Advanced Workflow

Git Flow Workflow



How Git Flow Works



- The Git Flow workflow uses a central repository as the communication hub for all developers.
- Developers work locally and push branches to the central repo.

Historical Branches

- Instead of a single `master` branch, this workflow uses two branches to record the history of the project.
  - The `master` branch stores the official release history
  - The `develop` branch serves as an integration branch for Features
  - You should also tag all commits in the `master` branch with a version number

Feature Branches

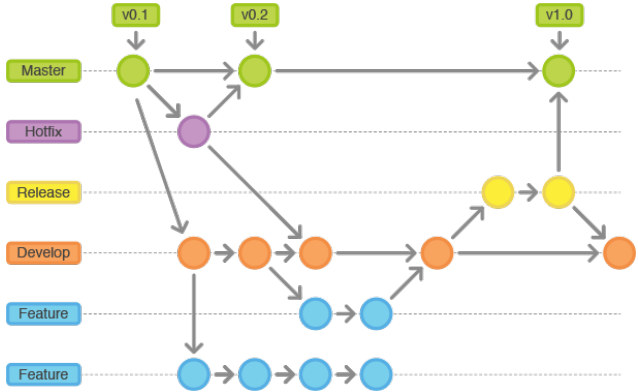
- Each new feature should reside in its own branch
  - Which is pushed to the central repo for backup/collaboration
  - `develop` is the parent branch for feature branches
  - Upon completion a feature branch is merged into `develop`
  - Features should never interact directly with `master`

Feature Branches - Best Practices

- May branch off: `develop`
- Must merge back into: `develop`
- Branch naming convention: anything except:
  - `master`
  - `develop`
  - `release-*`
  - `hotfix-*`

Release Branches

Maintenance Branches



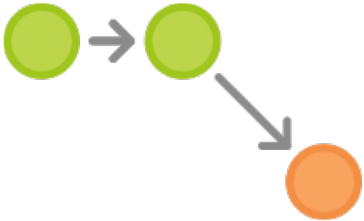
- Used to quickly patch production releases
- Upon complete it is to be merged both into `master` and `develop`

Maintenance Branches – Bests Practices

- May branch off: `master`
- Must merge back into: `master` and `develop`
- Tag: increment `patch` number
- Branch naming convension: `hotfix-*` or `hotfix/*`

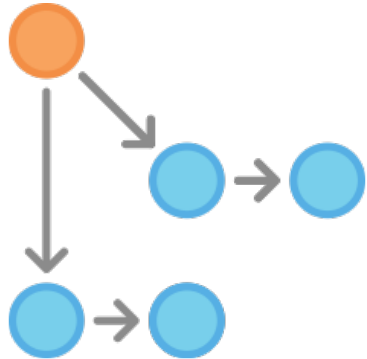
Git Flow Example

Create A Develop Branch



- Complement `master` with a `develop` branch locally and push it to the server.
- `develop` contains the project history, `master` contains an abridged version
- New developers should clone `develop` rather than `master`

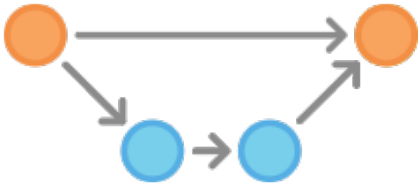
Beginning New Features



- Each developer should create a `feature` branch off of `develop`

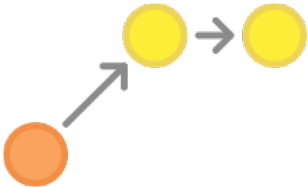
Git Flow Example

Finishing a Feature



- Once a feature is complete, the branch owner should either
  - make a pull request to have the branch merged with `develop`
  - or, merge it with their local copy of `develop` and push to the central repository

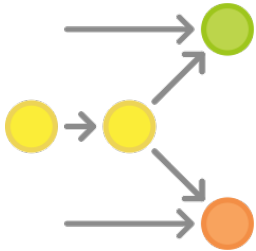
Preparing a Release



- Once ready to create a release, a new `release` branch off of `develop` should be created and named using Semantic Versioning
- The allows for cleanup of the release
- When ready it needs to be pushed to the central repository, where it becomes **feature-frozen**

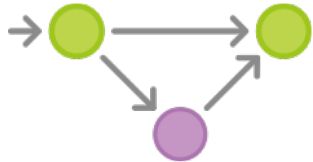
Git Flow Example

Finishing a Release



- Once ready to ship the `release` branch should be merged with both `master` and `develop`, and then it should be deleted.
- This is a great point at which to conduct a code review.
- At this point `master` should be tagged with the release version number

End-User Discovers a Bug



- End-user opens a ticket about a bug in the current release.
- To address this a new maintenance branch, aka `hotfix`, off of `master` is created
- Fixes are added and committed to the new branch and when fixed the branch is merged back into `master`
- `master` is tagged at this point with a version number updated by incrementing the `patch` number
  - v0.1.0 -> v0.1.1

Idaho State University
Computer Science

Git Flow Activity

1. Let's first create the `develop` branch
2. Now each of you checkout the `develop` branch and create a new `feature` branch

```
$ git checkout -b feature_name develop
$ git push -u origin feature_name
```

3. add a file named "yourname.txt" with your name in it.

```
$ git add .
$ git commit -m "message"
$ git push
```

4. Finish the feature

```
$ git pull origin develop
$ git checkout develop
$ git merge --no-ff feature_name
$ git push origin develop
```

5. Delete feature branch

```
$ git push origin --delete feature_name
```

ROAR

Are there any questions?