

Version Control

The backbone of Agile development

First, what is Version Control?

- A) Emailing a .zip of last night's code to your team after making changes to the .zip they sent you?
- B) Creating a .backup copy of every file you change in unfamiliar code, just in case?
- C) Checking your code into a central repository?
- D) The undo button?

E) All of the Above!

(to some extent)

- Version Control is whatever method you use to track changes to your code!
- Some key features/capabilities of version control:
 - Revert changes
 - View change history
 - Ability to reproduce an exact instance of a particular build
- Most if not all software companies use some standard* method of version control
 - *Most standard methods do not include zip files or emailing code

Version Control Tools (and why we love them!)

- GIT

- A crowd favorite, widely used among several tech companies in SLO and Silicon Valley
- Known for particularly useful collaborative features such as “git branch” and “git blame”

- SVN

- Similar to GIT in basic practice, but lacks some specific functionality that is widely adopted as part of the Agile workflow
- Can still be used effectively in Agile processes

- MERCURIAL

- Another form of version control I am less familiar with
- Someone somewhere uses it and loves it :)

* Personal anecdote: My first job used SVN and switched to GIT. Every company I have worked with since that point uses GIT exclusively

GIT: How it works

All project code is hosted in the remote “repository” typically on a cloud server or hosting service (more on these later)

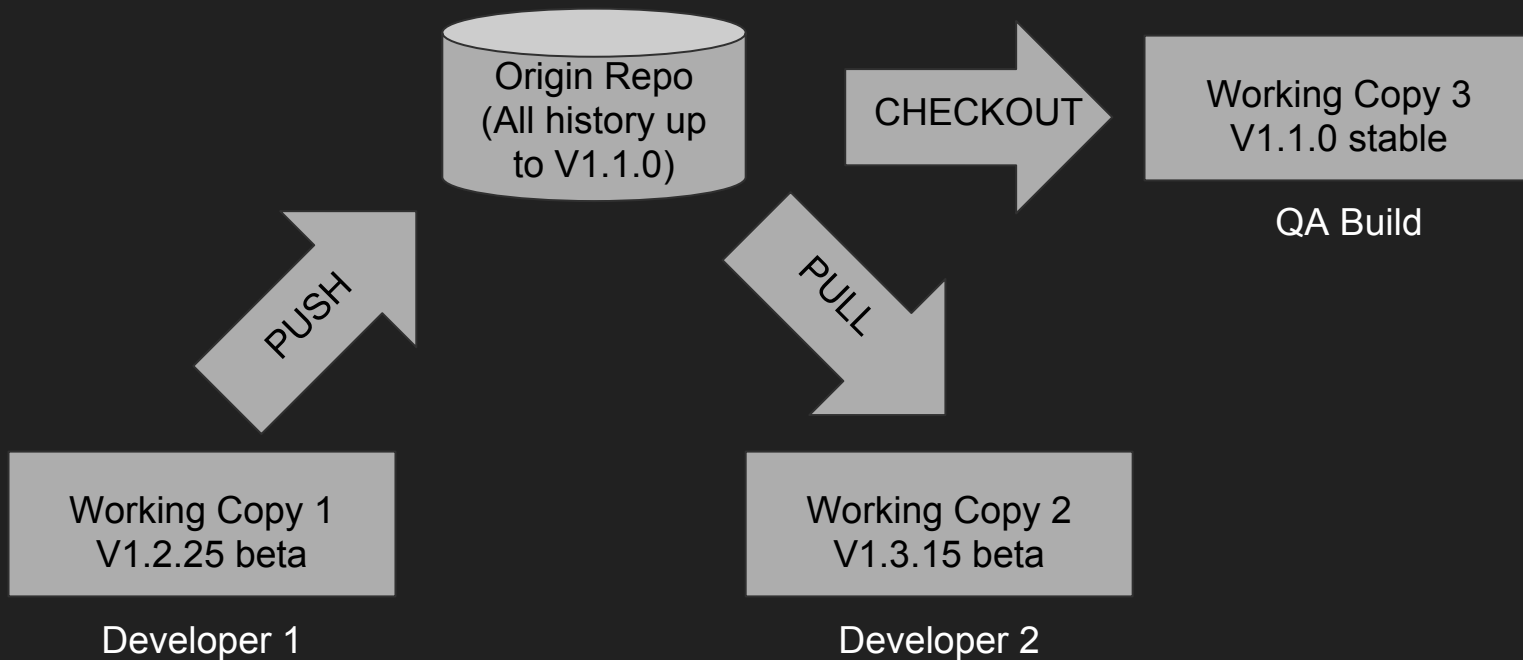
Users “checkout” a working copy of the code from the repository “origin” and store it on their local development machine

Code changes are submitted as a “commit” with a title and message. It is important that the commit messages are descriptive, accurate, and concise

Commits are “pushed” to the origin from one developer’s working copy, and can be “pulled” to other development machines tracking the same repository

The “git log” is a history of all commits. For a quick chuckle, feel free to reference <http://www.commitlogsfromlastnight.com> :)

GIT: The Block Diagram



Which version was that again?

The repository contains all commit history for that git project, including all versions of the code, all commit messages in each version, who submitted the commit, timestamp, etc.

These versions are typically tracked by a “branch” in the repository, and developers can “checkout” any branch they wish to view/edit

Your local working copy only retains enough information to display the current branch and allow the user to checkout other branches

Note: The `git` command is actually just a program on your computer that communicates with the remote repository and updates your local working copy accordingly. XCode has git built in, but we will focus on command line for today

Branch Management

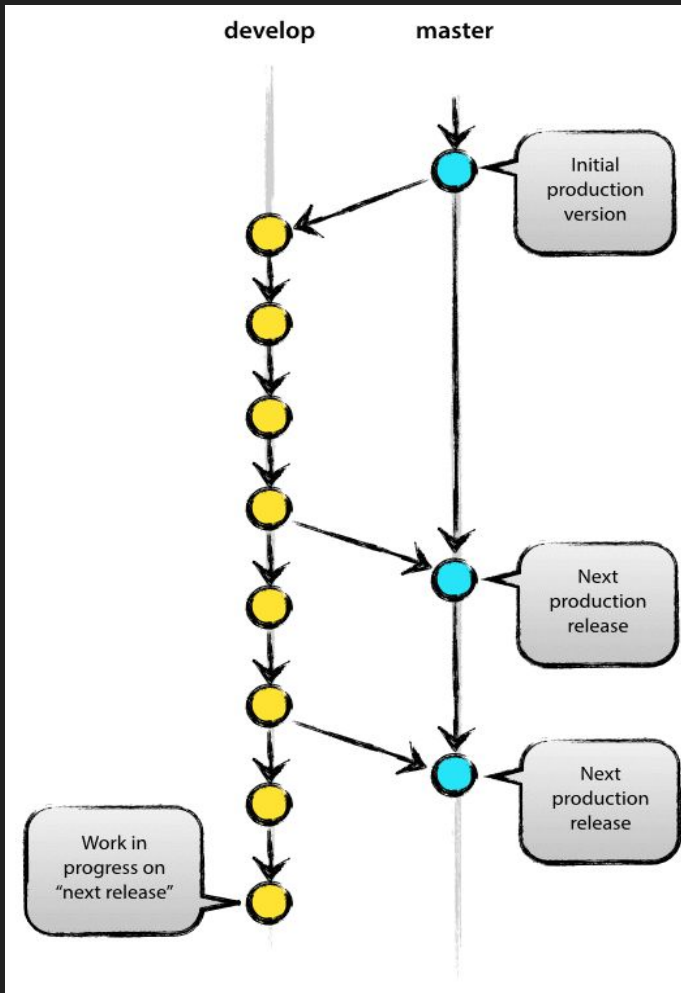
When checking out a branch, the code in your working copy changes to reflect the new version

Branches are useful for keeping “in progress” code separate from stable code

Branches can be merged into other branches or branched off further

Developers can checkout a branch, or a specific commit from a branch's history

ProTip: Checking out an older version of the code is often useful for debugging



GIT + Agile: Feature-Branches

Agile methodologies are intended to minimize the overhead associated with making changes to the project and roadmap.

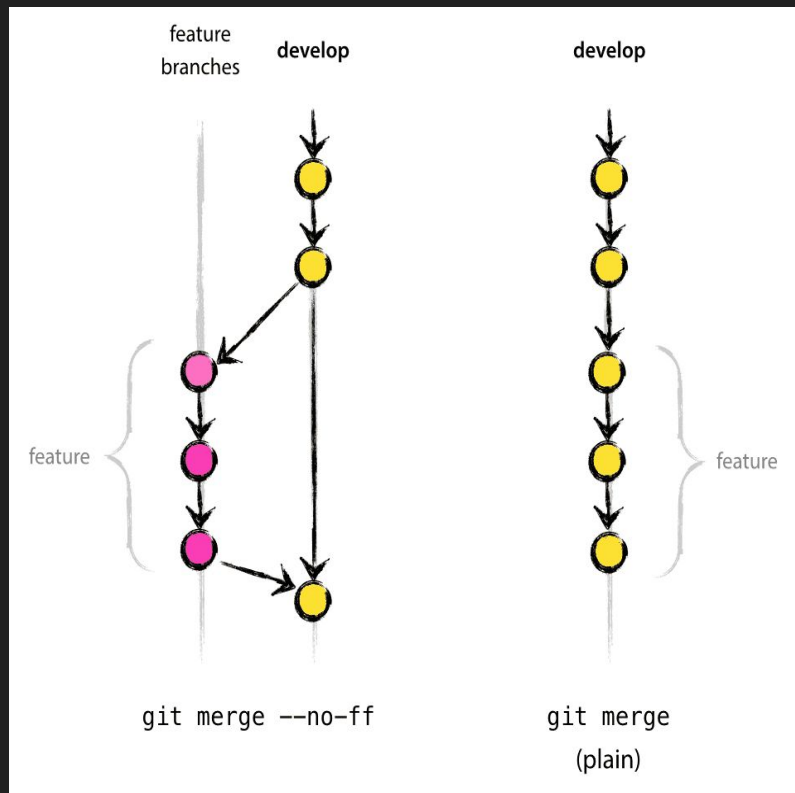
Aside from standing in a circle and talking every day, this usually means breaking down features into smaller tasks known as “Stories”

Stories happen to pair quite well with git branches, yielding a useful tool we refer to as the “feature-branch”

Feature-Branch: How we do it

Feature branches typically:

- Split off from the most recent stable version of the code
- Keep separate until the story/feature is complete or considered stable
- Allow development teams to work in parallel on separate tasks simultaneously
- Prevent untested or incomplete code from finding its way into production



The Pull Request

Repository hosting sites often include supplementary features to enhance version control tools and even extend the functionality

GitHub, BitBucket, and other hosts provide web GUIs for viewing code, comparing differences between branches, and other useful actions


The most useful, in my opinion, is the “Pull Request”

When a developer is ready to submit the new feature to the stable codebase, a pull request is created, providing a visual “diff” of the target branch vs the changes in the new feature branch

What's the Diff?

Code diffs are primarily used for Code Review before accepting the request and pulling the new version of code into the target branch

It highlights the changes in the Pull Request, and it provides a platform to discuss problems or alternative solutions

		@@ -24,11 +24,14 @@ - (void)dealloc {
24	24	[NotificationCenter removeObserver:self];
25	25	}
26	26	
27		-- (instancetype)initWithCoder:(NSCoder *)aDecoder {
28		- self = [super initWithCoder:aDecoder];
	27	+ (instancetype)init {
	28	+ self = [super init];

Merge: Bringing it all together

When a feature is complete, meaning it has been self-tested and passed code review from a peer or supervisor, the feature-branch is ready to merge!

Merging branches effectively evaluates all the commits in the feature branch that aren't in the target branch, then adds the commits to the target branch in a “merge commit”

The merge action will attempt to combine all changes cleanly and result in the latest version of code from all merged branches

Merges are usually capable of completing automatically (a single button press on the Pull Request in GitHub), but there is an exception...

Conflicts

Unfortunately, git doesn't always know which version of code is the latest, specifically when the same line of code is changed in multiple branches

When this happens, the merge will pause before it is completed, providing the developer with the opportunity to resolve the conflict manually and complete the merge commit

```
CONFLICT (content): Merge conflict in MercedesMe/Dealers.storyboard
ussalrmg4822:mercedesme-ios derhalma$ git status
On branch 2.6.2-develop
Your branch is up-to-date with 'origin/2.6.2-develop'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   MercedesMe/CustomerSupportMenu.m
        modified:   MercedesMe/MyProfile.storyboard

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

        both modified:   MercedesMe/Dealers.storyboard
```

Avoiding Conflict

Sometimes conflicts are unavoidable, and in those cases, it takes some care to resolve them properly (this is enough of a discussion for its own presentation)

In iOS projects, conflicts are most common when multiple branches commit changes to the same storyboard (due to extensive meta data managed by xcode), but they can happen in any type of file

The best way to avoid conflicts is to separate Stories/Subtasks such that it minimizes simultaneous changes in the same storyboard file or code function

This can sometimes mean creative management of storyboard references, or even replacing storyboards with programmatic ui methods if overlapping feature development cannot be avoided

Basic git commands (you'll use these daily)

``git checkout`` - switches your working copy to another branch

``git add`` - prepares some or all changes for a commit (more on this in lab)

``git commit`` - creates a history entry for this version of code, logs all changes to the code base. This can be considered the “save button”

``git pull`` - updates your current branch to the latest commit submitted to the repository. This can be considered the “refresh button”

``git push`` - updates the repository with all commits on your working copy that have not been submitted. This can be considered the “share button”

``git status`` - views the current status of your working copy, printing out:

- any changes from what is saved in the last commit
- any commits that differ from your local copy and the remote repository

Advanced git commands (when things go wrong)

``git blame`` - view current version of file line by line with annotations for what commit last changed each line. Useful in debugging or reverse engineering an inherited code base... or figuring out who caused that bug ;)

``git cherry-pick`` - pull 1 commit from a branch to another, useful when separating features out of a given branch or including dependencies from another feature

``git bisect`` - binary search through versions of commit history to determine which commit affected a given observed behavior, useful in debugging recent changes that already made it into the stable branch

``git rebase`` - sometimes used as part of a standard merge workflow, this is used to “rewrite history” as a complex method of resolving conflicts. As a personal preference, I tend to avoid this whenever possible

In Summary

Git repositories act as a robust codebase backup and a platform for collaborative development

GitHub extends this functionality with cloud hosting and additional tools for managing projects and teams

Your local working copy can pull changes from the repository, push changes to the repository, or checkout different versions saved in the repository

Feature branches are used to help manage project maintenance

Pull Requests act as gates for features to enter the stable version

Conflicts arise when git is unable to tell which branch contains the most recent version of a file

Thank you!

Questions?