Primer for Git

[Tip 1: Start thinking about tasks as Git branches] [Tip 2: Multiple branches are individually testable, so take advantage] [Tip 3: Git provides transparency and quality to agile development] [Release branching] [Feature branching] [GitHub 'hello world' tutorial] [Training videos] [Cheat sheet] [Git champions]

Git is the de facto standard for agile software development when it comes to version control systems. This well-supported open source project is flexible enough to support a range of workflows that suit the needs of any given software team. Its distributed -- rather than centralised – nature gives it superior performance characteristics and allows developers the freedom to experiment locally and publish their changes only when they're ready for distribution to the team.

Besides the benefits of flexibility and distribution, there are key functions of Git that support and enhance agile development. Think of Git as a component of agile development; changes can get pushed down the deployment pipeline faster than working with monolithic releases and centralised version control systems. Git works the way your agile team works (and should strive to work).
Pro tip: Git is Distributed Version Control System (DVCS). Unlike CVS or Subversion (SVN) repositories, Git allows developers to create their own, personal copy of the team's repository, hosted alongside the main codebase. These copies are called forks and when work is complete on a fork, it's easy to bring changes back to the main codebase.

Tip 1: Start thinking about tasks as Git branches

Git comes into play after features have been fleshed out, added to a product roadmap, and the development team is ready. Once you have your tasks broken down into user stories then Git starts fitting into your agile workflow at this point. Ideally we should create a new branch for every single issue - Whether it's a new feature, a bug fix, or a small improvement to some existing code, every code change gets its own branch.

Branching is straightforwards and allows teams to easily collaborate inside one central codebase. When a developer create a branch, they effectively have their own isolated version of the codebase in which to make changes. For an agile team this means that by breaking features into user stories and then branches, the development team has the ability to tackle tasks individually and work more efficiently on the same code but in different repositories; there is no doubling up of work and since individuals are able to focus on small chunks of work in repositories separate from the main repository there are not as many dependencies slowing down the development process.
Pro tip: There are other types of Git branching besides this task branching (we call it 'feature branching') and they aren't mutually exclusive. You can create branches for a release, for example. This allows developers to stabilise and harden the work scheduled for a particular release, without holding up other developers who are working on future releases.

Tip 2: Multiple branches are individually testable, so take advantage

Once branches are considered done and ready for code reviews, Git plays another key role in an agile development workflow: testing. Successful agile teams practice code reviews and setup automated tests (Continuous Integration or Continuous Development). To help with code reviews and testing, developers can easily notify the rest of their team that the branch work is ready for review and that it needs to be reviewed through a pull request. More simply put, a pull request is a way to ask another developer to merge one of your branches into the master branch and that it is ready for testing.

With the right tooling, your Continuous Integration server can build and test your pull requests before you merge them - This gives you confidence that your merge won't cause problems. This confidence makes it easier to retarget bug fixes and conflicts in general, because Git knows the difference between the branch and master code base since the branches have diverged.
Pro tip: A long running feature branch that is not merged to the master branch may hurt your ability to be agile and iterate. If you have a long running feature branch remember that you effectively have two divergent versions of your code base, which will result is more bug fixes and conflicts. But the best solution is to have short-lived feature branches. This can be achieved through decomposing user stories into smaller tasks, careful sprint planning, and merging code early to ship as dark features.

Tip 3: Git provides transparency and quality to agile development

The Git/agile story is one about efficiency, testing, automation, and overall agility. Once you've merged a branch to the master branch, your agile workflow is done. Likewise, merging code through pull requests means that when code is done, you have the documentation to confidently know that your work is green, that other team members have signed off on the code, and that it is ready to release. This keeps agile teams moving at a speed and with confidence to release often: a sign of a great agile team.
Pro tip: Adopting a regular release cadence is key to agile development. In order to make Git work for your agile workflow, you need to make sure that your master is always green. This means that if a feature isn't ready then wait for the next release. If you practice shorter release cycles this shouldn't and won't be a big deal.

A free and quite good GUI for Git is SourceTree

https://www.sourcetreeapp.com/

You need to get comfortable with the concepts of:

- Cloning a repo
- Staging and committing changes
- Branching and merging

- Fetching and pulling
- Pushing to your origin repo on github.com
- Managing multiple remotes(origin, upstream, contributors' forks, etc.)

It really helps to get confortable with the git command line while learning Git, instead of completely relying on the SourceTree interface. You'll need to know your way around the command line if you want to leverage the full power of Git with more advanced features like the stash, rebasing and cherry-picking. Create some test repositories to play around with the various features before trying to contribute.

Release branching

Release branching refers to the idea that a release is contained entirely within a branch. This means that late in the development cycle, the release manager will create a branch from the master (e.g., “1.1 development branch”). This should be done when all the code is checked in and tested.
Warning: Release branching is an important part of supporting versioned software out in the market. A single product may have several release branches (e.g., 1.1, 1.2, 2.0) to support sustaining development. Keep in mind that changes in earlier versions (i.e., 1.1) may need to be merged to later release branches (i.e., 1.2, 2.0).

Feature branching

A feature branch model keeps all of the changes for a particular feature inside of a branch. When the feature is fully tested and validated by automated tests, the branch is then merged into Develop (and eventually into Master).

Feature branches are often coupled with feature flags–"toggles" that enable or disable a feature within the product. That makes it easy to deploy code into Master and control when the feature is activated, making it easy to initially deploy the code well before the feature is exposed to end-users.
ProTip: Another benefit of feature flags is that the code can remain within the build but inactive while it's in development. If something goes awry when the feature is enabled, a system admin can revert the feature flag and get back to a known good state rather than have to deploy a new build.

GitHub 'hello world' tutorial

Learn how to create a GitHib account, create a repo, create a branch, make a commit, open and merge a pull request:

https://guides.github.com/activities/hello-world/

Training videos

- Webcast - The Basics of Git and GitHub (51min)
- Interactive exercise - Learn Git in 15 minutes

You can also find a longer video series of small videos (3-5 minutes per video) going through specific topics, like, commit, forking, diff etc. All of them are gathered below:

1. Introduction
2. Setup
3. Config
4. Init
5. Commit
6. Diff
7. Log
8. Remove
9. Move
10. Ship of Theseus
11. Ignore
12. Branch
13. Checkout
14. Merge
15. Network
16. GUI
17. Intro to GitHub
18. Forking
19. Pull Requests

20. Reset
21. Reflog
22. Rebase

Cheat sheet

A handy downloadable GitHub cheat sheet

Git champions

There are a few people dotted about the place who have put themselves forward as 'Git Champions'. These are helpful souls who have experience with Git, and will be happy to help out others who are struggling or generally having some difficulties. If you are having trouble then contact your nearest Champion, but do try to keep queries outside of the 10:00 to 15:00 core hours.

- Manjunath Nayak <Manjunath.Nayak@river-island.com>
- Saroja Raut <Saroja.Raut@river-island.com>
- Keith Devlin <keith.devlin@river-island.com>
- Maryl Rodrigues <Maryl.Rodrigues@river-island.com>
- Sarika Sharma <Sarika.Sharma@river-island.com>