# Separating Collated Code with Branching Strategies

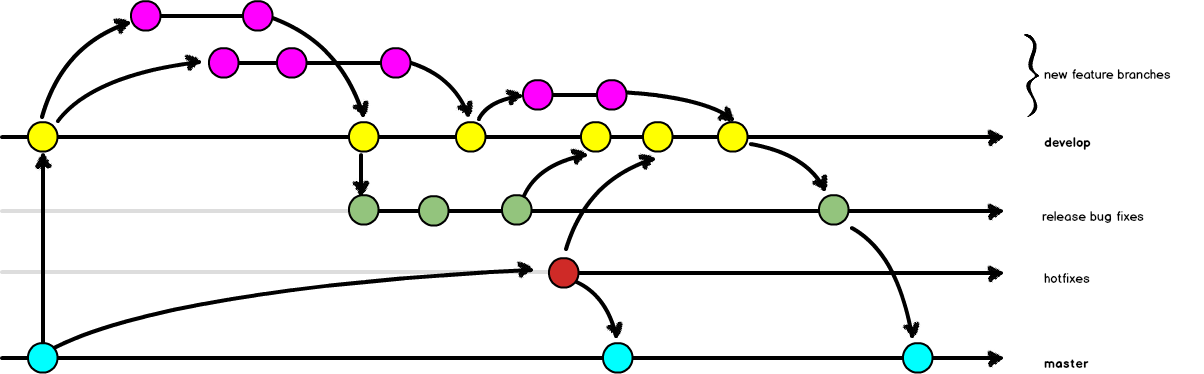
Identify and describe how your code is collated within your repository

# Branching Strategies

* Scheduled Deployment: Gitflow [http://nvie.com/posts/a-successful-git-branching-model/] Simplified Gitflow [http://drewfradette.ca/a-simpler-successful-git-branching-model/]
* Branch Per Feature [https://www.acquia.com/blog/pragmatic-guide-branch-featuregit-branching-strategy] or GitHub Flow [http://scottchacon.com/2011/08/31/github-flow.html]
* State Branching GitLab Flow [https://about.gitlab.com/2014/09/29/gitlab-flow/]

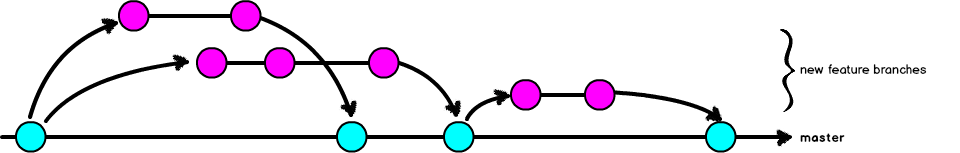
# Scheduled Deployment

* Optimized for the collation of many smaller changes into a single release.
* Typically used for a download-able product; or web site with a scheduled release cycle (e.g. "Wednesdays").
* Incorporates human-reviews, and possibly automated tests



# Branch-per-Feature Deployment

* Code is deployed faster than scheduled releases; assumes all check-ins are deployable.
* Requires (trusted) test coverage.
* Typically uses a mechanical gatekeeper (CI) to check in code to the master branch.
* Often has flippers/flags for fine grained access to in-progress features.
* Fewer branches to maintain / keep updated.



Note: if you don't need the granularity of multiple supported versions, you can probably get away with something closer to this branching strategy. Can you get away with just tags? Do you intend to go back and work on a previous version? As soon as you have the concept of a separate security hotfix, you need to introduce a separate branch. In CD: everything is urgent, so there's not a separation of a really urgent security fix. CI, CD vs CD: http://puppetlabs.com/blog/continuous-delivery-vs-continuous-deployment-whats-diff

# Creating and maintaining branches

NOTE: you must keep your branch up with dev. At least daily you should update and merge the dev branch into your working branch(es). If you don't, we can end up with PITA conflicts when moving back into master and live. Any tickets that have merge conflicts will be reopened and kicked back to you to fix before review.

**To make a branch:**

* git checkout dev
* git pull origin dev
* git branch <branchname>
* git checkout <branchname>
* git push origin <branchname>

Do your work and push from here.

**To keep your branch current with dev:**

* git checkout dev
* git pull origin dev
* git checkout <branchname>
* git rebase dev

**Conducting a Peer Review**

Checkout a local copy of the branch relating to the tick

* git branch -a # show a list of all available branches
* git fetch # downloads the branches
* git checkout --track -b NNNN-branch-name origin/NNNN-branch-name # NNNN should be the ticket number

(short version of above = git checkout --track origin NNNN-branch-name)

Once you've confirmed the branch is correct and complete it can be merged into the dev branch. Ensure your branch is up-to-date:

* git checkout dev
* git pull
* git checkout NNNN-branch-name
* git rebase dev

Merge the branch you've reviewed into dev and push it back up.

* git checkout dev
* git merge --no-ff NNNN-branch-name
* git push

Delete the ticket branch as follows:

* git branch -d NNNN-branch-name # delete the local branch
* git push origin --delete NNNN-branch-name # delete the remote branch

Delete the ticket branch as follows:

* git branch -d NNNN-branch-name # delete the local branch
* git push origin --delete NNNN-branch-name # delete the remote branch

Note that you can do git fetch -p origin. The -p option tells fetch to delete any tracking branches that no longer exist in the corresponding remotes; by default they are kept around.

# Article https://www.derekgourlay.com/blog/git-when-to-merge-vs-when-to-rebase/

Keeping a clean history in git comes down to knowing when to use merge vs. Rebase

### Rule of thumb:

* When pulling changes from origin/develop onto your local develop use rebase.
* When finishing a feature branch merge the changes back to develop.

### Use git pull --rebase when pulling changes from origin

Use git merge when finishing off a feature branch.

While it is possible to set your system to default to git pull --rebase over using the regular git pull you will occasionally find you run into problems such as the following scenario:

The problem is…

### Rebasing Deletes Merge Commits!

### Welcome the –preserve-merges flag to center stage:

From the git-rebase manpage:

−p, −−preserve−merges

Instead of ignoring merges, try to recreate them.

This uses the −−interactive machinery internally,

but combining it with the −−interactive option explicitly

is generally not a good idea unless you know what you are

doing (see BUGS below).

So, instead of using git pull –rebase, use:

### git fetch origin and git rebase -p origin/develop

Unfortunately the -p flag cannot be used in conjunction with git pull ( git pull –rebase -p doesn’t work!) and as a result you have to explicitly fetch & rebase changes from origin.

**ORIG\_HEAD is no longer preserved**

Unlike git pull –rebase, which will fetch changes from the branch your current branch is tracking, git rebase -p doesn’t have a sensible default to work from. You have to give it a branch to rebase from (which is why we specify origin/develop in the above example).

To avoid messy merge commits and help keep a relatively clean git commit history use the following workflow when fetching upstream changes:

### git fetch origin

### git rebase −p origin/develop

Why should I rebase ?

* Because it makes your commit history easier to read and manipulate

When should I rebase ?

* Before sharing proposed work
* To update your local working branch
* To split a commit into two