Code development workflow and good practices

J. Sturdy

Wayne State University

May 15, 2018

GEM Online/Commissioning Meeting

Why a manifesto?

- Collaborative effort means working with people who may be doing things differently
  - Different text editors sometimes treat "basic" things differently, be aware of this
  - Different operating systems treat line endings differently

Example

- newline at end of file
- CRLF vs LF
- Before adding/committing a change, do a `git diff`

$\Rightarrow$ Goal: Minimize the effects from collaboration with people using the tools they are comfortable with

Why a manifesto?

- Collaborative effort means working with people who may be doing things differently
  - Different text editors sometimes treat "basic" things differently, be aware of this
  - Different operating systems treat line endings differently
- We have to work to ensure that the differences in style and mechanics do not result in a difficult to understand and document codebase, while at the same time allowing developers the freedom to develop code in a way that is comfortable for them

$\Rightarrow$ Goal: Minimize the effects from collaboration with people using the tools they are comfortable with

C/C++ style

- Have loosely been following the Google style guide `https://google.github.io/styleguide/cppguide.html`
  - Slowly adapting all our code to adhere to several of the guidelines

$\Rightarrow$ Goal: Minimize the differences in style from different parts of the code so the uniformity of the codebase is maintained

`git` workflow

- All this style and semantics aside, at the crux, collaborative development requires cooperation
- Our development is primarily based on the `git-flow` workflow
  - Primary "central" repository

- Developers fork the central repository to their own `github`/`gitlab` account
- Pull requests to central when features have been developed
- Pull requests between developers for specific new features not yet pushed to main repository

$\Rightarrow$ Goal: Coherent and cohesive development environment

`git` workflow

- All this style and semantics aside, at the crux, collaborative development requires cooperation
- Our development is primarily based on the `git-flow` workflow
  - Primary "central" repository
  - Primary (protected) branches in the central repository
    - `master`
    - `develop`

- Branches that are central should not be committed to directly, and should not be merged in a fork unless special care is taken to avoid extraneous "merge commits"
- `master` and `develop` are primary and protected
- `master` should **always** be "stable", and all (most?) release tags will exist on `master`
- `develop` will be the initial branch point for all future `release-`s
- `develop` branch should be the branch point for (almost) all `feature-`s
- `release-<rel-ver>` will be protected
- If your fork has these branches, there should be **no** difference between them and the central repository

`git` workflow

- All this style and semantics aside, at the crux, collaborative development requires cooperation
- Our development is primarily based on the `git-flow` workflow
  - Primary "central" repository
  - Primary (protected) branches in the central repository
    - `master`
    - `develop`
    - `release-<rel-ver>`

- `release-<rel-ver>`
  - Is a protected branch, used as an integration area for features targeting a new release
  - Is branched off of `develop` once a release is imminent and new features in this release are frozen
  - Specific features **may** get their own branch from `release-`

⇒ Goal: Coherent and cohesive development environment

`git` workflow

- All this style and semantics aside, at the crux, collaborative development requires cooperation
- Our development is primarily based on the `git-flow` workflow
  - Primary "central" repository
  - Primary (protected) branches in the central repository
    - `master`
    - `develop`
    - `release-<rel-ver>`
  - Secondary branches (possibly) in the central repository, probably in your fork
    - `feature-<some feature>`

- Included into appropriate upstream branch via pull request
- `feature-` branches are for starting development of a new feature to be added to the code, they are made off of the `develop` branch

`git` workflow

- All this style and semantics aside, at the crux, collaborative development requires cooperation
- Our development is primarily based on the `git-flow` workflow
  - Primary "central" repository
  - Primary (protected) branches in the central repository
    - `master`
    - `develop`
    - `release-<rel-ver>`
  - Secondary branches (possibly) in the central repository, probably in your fork
    - `feature-<some feature>`
    - `hotfix-<some hotfix>`

- Included into appropriate upstream branch via pull request
- `feature-` branches are for starting development of a new feature to be added to the code, they are made off of the `develop` branch
- `hotfix-` branches are for specific bugs discovered in `master`

`git` workflow

- All this style and semantics aside, at the crux, collaborative development requires cooperation
- Our development is primarily based on the `git-flow` workflow
  - Primary "central" repository
  - Primary (protected) branches in the central repository
    - `master`
    - `develop`
    - `release-<rel-ver>`
  - Secondary branches (possibly) in the central repository, probably in your fork
    - `feature-<some feature>`
    - `hotfix-<some hotfix>`
    - `bugfix-<some bugfix>`

- Included into appropriate upstream branch via pull request
- `feature-` branches are for starting development of a new feature to be added to the code, they are made off of the `develop` branch
- `hotfix-` branches are for specific bugs discovered in `master`
- `bugfix-` branches are for specific bugs discovered in other protected branches, and should be created appropriately

Hotfix example

- Suppose a bug is found in some current "stable" tag

$\Rightarrow$ Goal: Fix specific known problem and reintegrate as quickly as possible with minimal extraneous changes

Hotfix example

- Suppose a bug is found in some current "stable" tag
  - Obtain the up-to-date master branch

Command examples

```
git remote add gemdaq git@github.com:cms-gem-daq-project/<repo>.git
git checkout -b gemdaq-master gemdaq/master
```

⇒ Goal: Fix specific known problem and reintegrate as quickly as possible with minimal extraneous changes

Hotfix example

- Suppose a bug is found in some current "stable" tag
  - Obtain the up-to-date master branch
  - Create a new branch "hotfix-some-bug"

Command examples

```
git checkout -b hotfix-some-bug
```

⇒ Goal: Fix specific known problem and reintegrate as quickly as possible with minimal extraneous changes

Hotfix example

- Suppose a bug is found in some current "stable" tag
  - Obtain the up-to-date master branch
  - Create a new branch "hotfix-some-bug"
  - Find and fix the bug modifying **nothing else**

$\Rightarrow$ Goal: Fix specific known problem and reintegrate as quickly as possible with minimal extraneous changes

Hotfix example

- Suppose a bug is found in some current "stable" tag
  - Obtain the up-to-date master branch
  - Create a new branch "hotfix-some-bug"
  - Find and fix the bug modifying **nothing else**
  - Push the hotfix branch to github

Command examples

```
git commit -m "fixed #N"
git push --set-upstream origin hotfix-some-bug:hotfix-some-bug
```

⇒ Goal: Fix specific known problem and reintegrate as quickly as possible with minimal extraneous changes

Hotfix example

- Suppose a bug is found in some current "stable" tag
  - Obtain the up-to-date master branch
  - Create a new branch "hotfix-some-bug"
  - Find and fix the bug modifying **nothing else**
  - Push the hotfix branch to github
  - Create a pull request to the central `master`

⇒ Goal: Fix specific known problem and reintegrate as quickly as possible with minimal extraneous changes

Hotfix example

- Suppose a bug is found in some current "stable" tag
  - Obtain the up-to-date master branch
  - Create a new branch "hotfix-some-bug"
  - Find and fix the bug modifying **nothing else**
  - Push the hotfix branch to github
  - Create a pull request to the central `master`
  - Create a pull request to the central `develop` or current `release-` branch (maintainers should perform the appropriate `cherry-pick`/`patch` application)

⇒ Goal: Fix specific known problem and reintegrate as quickly as possible with minimal extraneous changes

Feature development example

- Decide to develop "cool new feature" for some future release

$\Rightarrow$ Goal: Isolate different features and the code required to develop them, streamlines the simultaneous parallel developments by different people on different features

Feature development example

- Decide to develop "cool new feature" for some future release
- Branch off of `develop`

Command examples

```
git remote add gemdaq git@github.com:cms-gem-daq-project/<repo>.git
git checkout -b gemdaq-develop gemdaq/develop
git checkout -b cool-new-feature
```

⇒ Goal: Isolate different features and the code required to develop them, streamlines the simultaneous parallel developments by different people on different features

Feature development example

- Decide to develop "cool new feature" for some future release
- Branch off of `develop`
  - The feature should be compartmentalized as much as possible

$\Rightarrow$ Goal: Isolate different features and the code required to develop them, streamlines the simultaneous parallel developments by different people on different features

Feature development example

- Decide to develop "cool new feature" for some future release
- Branch off of `develop`
  - The feature should be compartmentalized as much as possible
  - Including changes from other concurrent developments is fine if they go through the `develop` branch, or the upstream protected branch from which your `feature-` branch was created

Command examples

```
git fetch -p --all
git checkout gemdaq-develop
git pull
git checkout cool-new-feature
git rebase gemdaq-develop
```

⇒ Goal: Isolate different features and the code required to develop them, streamlines the simultaneous parallel developments by different people on different features

Feature development example

- Decide to develop "cool new feature" for some future release
- Branch off of `develop`
  - The feature should be compartmentalized as much as possible
  - Including changes from other concurrent developments is fine if they go through the `develop` branch, or the upstream protected branch from which your `feature-` branch was created
- When finished, push your `feature-` branch and create a pull request to the appropriate upstream branch

$\Rightarrow$ Goal: Isolate different features and the code required to develop them, streamlines the simultaneous parallel developments by different people on different features

Release example

- `release-<rel-ver>` branches will be created only by repository maintainers when a new release is being targeted

Command examples

```
git checkout develop
git checkout -b release-1.2
```

⇒ Goal: Bring in all features up to a certain point to target a new stable release of the software

Release example

- `release-<rel-ver>` branches will be created only by repository maintainers when a new release is being targeted
- After the branch is made, feature branches merge into the release branch (done via pull requests)

Command examples

```
git merge --no-ff feature-1.2-cool-1
git commit -m "merging feature-1.2-cool-1 into release-1.2"
git merge --no-ff feature-1.2-cool-2
git commit -m "merging feature-1.2-cool-2 into release-1.2"
```

⇒ Goal: Bring in all features up to a certain point to target a new stable release of the software

Release example

- `release-<rel-ver>` branches will be created only by repository maintainers when a new release is being targeted
- After the branch is made, feature branches merge into the release branch (done via pull requests)
- When finalized, the `release-<rel-ver>` branch is merged into `develop`, and then into `master` and tagged

Command examples

```
git checkout develop
git merge --no-ff release-1.2
git commit -m "merging release-1.2 into develop"
git checkout master
git merge --no-ff release-1.2
git commit -m "merging release-1.2 into master"
git tag -a -m "tagging release-1.2 as v1.2.0" v1.2.0
```

⇒ Goal: Bring in all features up to a certain point to target a new stable release of the software

General contribution guidelines

- Do **not** `git rebase` a branch which you have pushed which others are now using
- Do **not** `git commit -a` without verifying that you haven't added unexpected or unnecessary files
  - Especially `don't` do this and subsequently `git push`

⇒ Goal: Behaviours that will make everyone's lives easier, list to be added to

Backup

Important Links

- github organization (if you want develop, subscribe, fork, and issue pull requests)
  `https://github.com/cms-gem-daq-project`
- gitlab organization (will probably migrate fully here at some point)
  `https://gitlab.cern.ch/groups/cms-gem-daq-project/`
- Based on the following workflow
  `http://nvie.com/posts/a-successful-git-branching-model/`

Updating github repositories to new structure

- gemdaq-testing (cmsgemos) and gem-light-dqm are now separate repositories
- To update your github to reflect the new behaviour do the following
  - Ensure that your gem-daq-code repository is up to date with the central gem-daq-code
    - This means that your develop, master, and release branches are concurrent
    - Any unmerged branches that have ongoing developments should be pushed to your gem-daq-code github repository
- Fork the new repositories from the central into your own github
- Clone your gem-daq-project repository somewhere

Example

```
git clone git@github.com:jsturdy/gem-daq-code.git cmsgemos
split-em-up.sh $PWD/cmsgemos gemdaq-testing git@github.com:jsturdy/cmsgemos.git js
git clone git@github.com:jsturdy/gem-daq-code.git gem-light-dqm
split-em-up.sh $PWD/gem-light-dqm gem-light-dqm
git@github.com:jsturdy/gem-light-dqm.git js
```

$\Rightarrow$ Goal: Create new github repository for each of the split repositories, while keeping commit history