Git Usage Guidelines

John Harwell

Last Updated 11/02/2017

Updated by Aaron Koenigsberg 10/26/2020

Contents

This document describes how we will be expecting you to use git throughout this course, but ESPECIALLY on the project.

1 Commit Sizing

You should have a relatively large number of smaller commits when working on either individual assignments or on the project. Waiting until you have everything working and then committing things all at once is NOT proper use of git or version control in general. Every time you:

- Complete implementation of a feature

- Make progress in debugging a feature

- Fix a bug

- Scaffold a class

- Fix up the style of the code

- Add/modify test code

- Update documentation

- Move/rename files

You should create a commit for whatever you did. Each of these tasks should generally have their own commit. This may seem pedantic, but it makes for fine-grained/easy stepping back through history to figure out what change broke something. In addition, if you have an "and" anywhere in your commit message, that probably means that the commit should be split into multiple commits.

2 Commit Messages

Below is a git commit message template that will help you to write good commit messages. You do not have to follow it, but doing so will ensure the quality of your commit messages, which will be helpful not only for grading purposes, but also for when you have to step back through your commit history to find a breaking change. If your messages are things like "in progress commit", "working on feature xxx", or "Fixed bug", that is not helpful to anyone including future you.

However, that does not mean that every time you forget to save before committing, made a typo, forget a semicolon somewhere, etc. that you MUST use this commit template. This template should generally be used for more substantial commits, as described above.

```
# Git commit message template

# Layout:
# <Title> (aka <type>(<scope>): <subject>)
# <body>
# <footer>
#
#
# Layout Explanation:
# <scope>   : whatever module(s) are affected
```

```
# <type>     : Must be one of the types below:
#
#              feat     (new feature)
#              fix      (bug fix)
#              docs     (changes to documentation)
#              style    (formatting, missing semi colons, etc; no code change)
#              refactor (refactoring production code; i.e. no functionality change)
#              test     (adding missing tests, refactoring tests;
#                        no production code change)
#              chore    (updating grunt tasks etc; no production code change)
#
# <subject>     : Think 'If applied this commit will...'
# <body>        : Description of what you did, explaining WHY this change was
#                 made.
# <footer>      : Provide links to any relevant tickets, articles, or other resources
#
# Add your commit message below here, following the guidelines above. Delete
# everything above your commit (i.e. through this line). git will ignore it if you
# don't anyway.
```

In addition, all git commits that are on feature branches (see Feature Branches) should refer to the issue associated with that feature branch. This will make the commit show up on github under the associated issue, making the code development process much more self documenting.

3 Example messages

Here are some examples of good git commit messages, using the template above from last semester (different project, but same idea).

3.1 Example 1

```
Fix(filter_manager): Create FlashPhoto, MIA derived classes from FilterManager

- Much cleaner to add only desired filters to panel.
- Also added a new constructor argument for MIA GUI/cmd line applications
  for the location of the stamp/marker (previously it was just hardcoded).
```

3.2 Example 2

```
Fix(iter3-base): When initializing, GLUI must be initialized before GLUT.

- There were also 2 copies of BaseGfxApp (no error because one was in alibrary,
  I think). Kept the one in libapp/, as that seemed more appropriate.
```

3.3 Example 3

```
In progress attempting to get MIA to compile.
```

```
- Everything but the command line interface compiles.
- May replace command line interface with boost...
- Added MIA to Makefile, so typing make in the root directory no longer  works
- MIA can use the same filter manager, undo manager, as FlashPhoto.
  However, it will need to derive from IOManager to accomodate the additional buttons.
```

3.4   Example 4

```
refactor(kernel,filter): Continued refactoring

- Started fixing m_ member declarations
- First pass for proper file header/section header comments
- Removed several instances of derived classes deleting members belonging to
  base classes
- Started cleaning up functions (getters, setters, etc.)
```

4   Setting "git commit" editor

Type this into your shell:

```
git config --global core.editor "EDITOR_NAME"
```

Where `EDITOR_NAME` can be one of [emacs, vim, geany, atom, nano gedit], or any other editor that is installed on the CS lab machines. If you are comfortable on the command line, I would recommend emacs, vim or nano. If you are not, then one of the other graphical options will work better.

Also, if you place the above template into `~/.gitmessage`, it will automatically load everytime on a git commit (not required, but useful).

Typing `git commit` will bring up whatever editor you've chosen with an empty commit message. I use emacs, so mine opened in emacs:

Here's what the commit screen might look like:

```
1    feature(create-task-hierarchy): Start scaffolding task hierarchy classes (#25).
2
3    - Does not compile
4    - Needed to provide a means to decompose logical tasks into discrete units (task
5      sequences) that can actually be run by the robots.
6    - Does not employ any design patterns at the moment.
7    []
8    # Please enter the commit message for your changes. Lines starting
9    # with '#' will be ignored, and an empty message aborts the commit.
10   # On branch feature/25-create-task-hierarchy
11   # Changes to be committed:
12   #       modified:   include/rcppsw/patterns/visitor/visitor.hpp
13   #       new file:   include/rcppsw/task_allocation/atomic_task.hpp
14   #       modified:   include/rcppsw/task_allocation/decomposable_task.hpp
15   #       renamed:    include/rcppsw/task_allocation/base_task.hpp -> include/rcppsw/task_allocation/logical_task.hpp
16   #       new file:   include/rcppsw/task_allocation/task_sequence.hpp
17   #
```

Notice in the screenshot above the blue text at the top, "feature(create-task-hierarchy) ..."
follows the format `<type>(<scope>):  <subject>` as described in the message template above.
It indicates the addition of a feature (i.e. new functionality) that impacts the "task-hierarchy"
module, and also provides a brief description. The first line of this message is what will appear in
the description of the Github directory listing (shown below) for each file that was modified during
this commit.

After commiting and pushing to git, the first line of the commit message is displayed on github
next to any files/directories that were changed because of it:

Notice the first line in the editor appears in `include/rcppsw`, which is the directory containing
the files that were modified.

5   How We Will Grade git commits

When grading git usage we will look at a combination of:

1. How many commits you have for a given iteration.

2. The general quality of the commits. We should be able to `clearly` trace your development path just by looking at your commit history.

Thus, adhering rigorously to the above template for EVERYTHING will likely violate #2, just as not adhering to the template for ANYTHING will. However, as stated before you don't have to use the template, but past semesters have shown that students that do almost always have a clear, concise, easy-to-follow commit history.

6   Branching Model

In a general git workflow, such as most you will encounter in industry, the `master` branch is not the general purpose development branch. Instead a number of other branches are used to perform development, and only pristine (i.e. working/release/etc) versions of the code are pushed to `master`. For this class, we will be using the following branching model, which is inspired by http://nvie.com/posts/a-successful-git-branching-model.

6.1   release (master, in most other projects)

This branch should:

- Always contain 100% functional, working code. There is nothing worse than cloning someone's open source project on github, only to find that the master branch does not compile or work.

- Only be pushed to for releases, and nothing else. Anytime someone clones your repository and gets your master branch, they should **always** get a working version of code (even if it is not the most up-to-date, which could live on your `devel` branch). For this class, this will be limited to the final submission for an iteration.

This branch always exists in a git repository, so you will never need to create it.

6.2   `devel`

This is the main development branch, and also where all feature branches branch off of/get merged into. It should:

- Almost never be committed to directly, unless it is something like fixing trivial compiler errors, whitespace issues, etc. For all consequential development work, a feature branch should be created off of `devel`

- Branch off of **release** at the start of a project, and never merge back into **release**, except for code releases.

To create this branch, do the following while in the directory for your repo:

```
git checkout master
git checkout -b devel
git push --all origin
```

This will push your newly created `devel` branch to github. You can verify that you are indeed on a new branch with:

```
git branch
```

The rest of this document will assume that you named your development branch `devel`, though you could have named it `integration` as well.

6.3   Feature Branches

These are the branches that should contain the actual development work. They should:

- Be named something specific (i.e. not just feature/phase2, but feature/phase2-widgetA-scaffolding). Generally, the more specific the better, although creating a branch called "fix/typo-error" is clearly overkill. Use your discretion.

- Usually only contain one or two commits (commits can be squashed if necessary). If you find yourself on a feature branch with many commits that is not necessarily a bad thing, as the feature you are working on might be complex enough to warrant multiple commits. However, if you find yourself wanting/needing to add/modify/remove code for things that are unrelated to whatever the feature branch that you are working on is supposed to contain (i.e. fixing a problem you found in WidgetB while your branch is for modifying WidgetA), please `restrain yourself`. Simply open an issue in github to track what you have found/want to do (See Github Issues), and continue on with the work of the feature branch.

- **Always** branch off of `devel` / `integration`, and **always** merge back into it.

- Be deleted after they have been merged back into `devel`.

6.3.1   Feature Branch Naming

All feature branches should have a descriptive name, as described above. However, it is also often convenient to group the branches into categories. You can use whatever categories you like, as long as they make sense. Here is a list of categories that you will often see on large scale projects, and what they are used for:

- feature - An actual feature (i.e. adding NEW functionality to the code).

- refactor - Refactoring existing code, possibly changing existing functionality in the process.

- bugfix - Fix a bug that was found in the code.

- style - Updating/fixing code style (i.e. making it Google style compliant, for example).

- doc - Adding/updating documentation for the code.

- chore - Doing miscellaneous grunt work, such as file moving, renaming, etc.

In addition to a category, all feature branches (of whatever category) should **ALWAYS** have an associated issue number included in the name, which references an issue on github that contains a detailed description of the branch and why it is necessary, what needs to be done on it, etc. For example, if I want to add a new feature "random number generation" to my code, and I have opened issue #387 on github about it, I might name my branch `feature/387-random-number-generation`.

You might argue that:

- Assigning each feature branch to a category is extra work

- Having long branch names like that is too much to type

- Putting the issue # in the branch name is unecessary, and does not add any benefit

However, what this process does is make it as clear and explicit as possible for both new developers coming on to a project, as well as future you, how the project development is organized, and how the code is logically layed out. This benefit far outweighs a little additional typing (which can be greatly reduced by auto completion anyway).

6.3.2 Creating a Feature Branch

To create a branch of this type, do the following from your repository directory. You **must** be on the development branch.
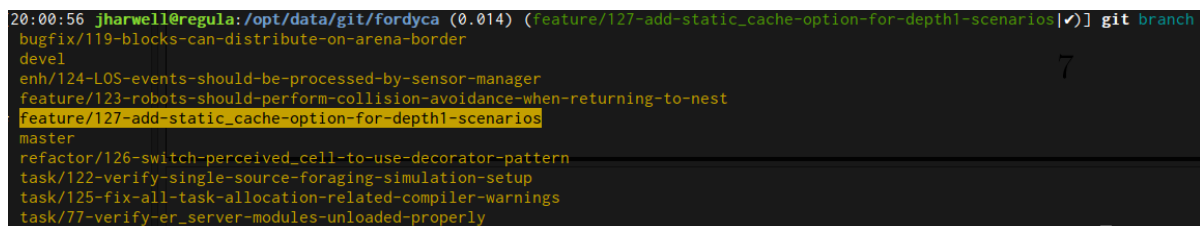
```
git checkout feature/123-my-awesome-feature
git push --all origin
```

Note: that you don't `have` to push all feature branches to github, though the example commands above do so. If you can keep things local, then you will generally be better off, and have less chance of messing up your repo.

Note: If a branch already exists on github, but not on your local repo, then running the first of the two commands above will get the remote branch and switch you to it on your local repo.

6.3.3 Examples of Feature Branches

Here are some examples of feature branches that I currently have for a project I'm working on:

8

6.3.4 Merging a Feature Branch into `devel`

Once you have finished working on a given task/feature/whatever then you should do the following to merge it back into `devel` (after you have finished your final commit on the branch).

```
git checkout devel
git merge --no-ff feature/123-my-awesome-feature
```

The `--no-ff` option tells git not to do a fast forward merge, and actually create a merge object, which is just a technical way of saying making it easier to roll back changes that break things.

6.3.5 An Example `devel` Merge History

Following this model, your `devel` merge history will look something like this:

Note the use of the github issue in both the branch name, as well as in the commit messages for branch.

6.3.6 Deleting a Merged Feature Branch

After merging a feature branch, you can delete it from your local repo by doing the following from the `devel` branch (or really any branch that is not the feature branch you are trying to delete):

```
git branch -d feature/123-my-awesome-feature
```

If you need to delete it from github as well, you can do:

```
git push origin :feature/123-my-awesome-feature
```

Note the : in the code above–it will not work without it.

If you have a bunch of branches on your local repo that don't exist on github, and you don't want/need them anymore (i.e. they have been merged), you can delete them en masse by doing:

```
git remote prune origin
```

7   Github Issues

As stated in Feature Branch Naming, all feature branches should have an associated issue on github. All issues created in github should:

- Have the **SAME** name as the feature branch the correspond to. Yes it can be a bit of typing, but you only have to do it once.

- Have all commits associated with the issue (that are on the corresponding feature branch) show up on the issue page. This can be done by referring to the issue # in the commit message with a #, (i.e. #139 for issue 139). See also Commit Messages.

- Contain a high level description (suitable for a project manager or developer generally familiar with how the code works, but not necessarily an expert in the specifics). The amount of description should be directly proportional to how important/complex whatever the issue is addressing is.

8   How We Will Grade Your Git Branching Model

When grading the branching model you chose to use for your project, we will be checking to see how closely you stuck to the guidelines outlined above. Most importantly, we will look at your git commit log from a branch point of view, and see how well it resembles the outline of the example above. For reference, a non-graphical version of commit log with branching information can be obtained via:

```
git log --oneline --graph --decorate --all
```

9   Tagging

9.1   Creating a Tag

It is possible that you will be asked to tag your final work for an iteration, in addition to pushing it to master. To do that you simply do the following (from the `master` branch in this case)

```
git tag release/iter1
```

In this case, I have tagged the first iteration. The name of the tag, `release/iter1` can be anything, but it should follow naming guidelines similar to those above (i.e. name it something descriptive).

9.2 Pushing a Tag

Once you have tagged something in your local repo, to push it to git, you can do:

```
git push --tags
```