Team Standards

09/27/18

Paired Planet Technologies
and
Lowell Observatory

Mentor:
Isaac Shaffer

Team:
Zach Kramer, Brian Donnelly, Matt Rittenback

The purpose of this team standards document is to establish a common understanding of expectations and facilitate efficient and effective collaboration.

Table of Contents

Team Members and Roles

Team Leader (Zach)

The **team leader** coordinates task assignments and ensures work is progressing, runs meetings, and makes initial efforts to resolve conflicts.

The team leader **is not** responsible for contributing more to tasks than other members and is mainly responsible for guiding them. It is each member's responsibility to contribute work, the team leader will just coordinate what each member should work on.

Webmaster (Matt)

The **webmaster** is in charge of creating and maintaining the team's website.

Customer Communicator (Zach)

The **customer communicator** coordinates and conducts customer communications.

Recorder (Matt)

The **recorder** maintains detailed meeting minutes and notes.

Architect (Brian)

The **architect** is primarily responsible for ensuring that core architectural decisions are followed during implementation. This does not cover the style guide. This covers a more conceptual approach to programming (e.g. "we should have a single class for X that does Y").

Release manager (Zach)

The **release manager** coordinates project versioning and branching, reviews and cleans up commit logs for accuracy, readability, and understandability, and ensures that any build tools can quickly generate a working release.

Coder (Everyone)

Every team member is a **coder**. Each individual will have their own role and speciality, though. This is difficult to specify without knowing the programming details of our project. We are unsure

if there will be a front-end application, for instance. We will use a scheme where each feature has a **lead developer** and 0 or more support developers.

The purpose of a lead developer is to take charge in the implementation of an idea that was agreed on by the group. This helps especially in other developers following his/her approach and style instead of everyone working independently. Every coder must conform to the style agreed upon in this document, and should follow the architecture layout that was decided on.

The lead developer will be tied to a feature. For instance, if there was front-end feature, one person could be the lead developer for it, and there could be another lead developer for implementing the Pov-Ray library. Then there could be a developer specializing in making the Pov-Ray library a sub-module while the lead works on part of the implementation. Development should be feature driven -- one step at a time.

The lead developer will be responsible for assigning sub sections of code to be programed, and then putting them together afterward. Once the program has been put together the architect needs to verify it, except where the architect is also the lead developer, in which case one of the other programers on that feature should verify it as a way to double check.

This gives everyone a chance to lead while not being too chaotic. With three coders this scheme may be less effective at times and we may drift from the scheme, but for the majority of the time we should follow this.

Every coder has a standard set of responsibilities:

- All code must follow the style guide. Code not following it will result in denied pull requests by the release manager.
- Every commit must be compile-able.
- It is the coder's job to commit meaningful messages and follow the message guidelines. The release manager's job is to clean up the commits afterwards (e.g. squashing them into larger commits). In the real world, every programmer is expected to do this themselves, but it is more efficient for one person to do it here.
- Every coder follows version control standards (e.g. creating and solving issues). It is not the job of the lead developer, release manager, or team lead.

Team Meeting Expectations

Meeting times

The place of meeting can rotate, but the default will be a study room at the Cline Library when there is no other scheduled place. We will meet at default once a week for a dynamically determined amount of minutes, based on the workload. The default time will be Monday at 1pm, but can also be changed depending on workloads as long as the minimum time per month is kept the same (4 hours).

Agenda structure

Meeting should always start with a quick recap (1 minute a person) of what was done since the last meeting, what your task is until the next meeting, and any issues that came up. After this brief recap, an agenda should be set and recorded by the recorder. These issues should then be addressed in order, and any issues that we didn't have time to discuss should roll over and be prioritized in the next meeting if they haven't been resolved.

Issues regarding coding and bugs should be of lower priority. Issues dealing with due dates and deliverables will be dealt with first. This will be followed by an issues revolving around team cohesion. Most of the meeting should be focused on strategies for solving a problem and architecture design.

It’s expected to keep in communication between meetings via Slack. GitHub will be the best place to post any bugs or direct concerns about code. The code in question should be referenced directly through GitHub, or at least made available so other members can replicate the problem.

At the end of every meeting, we will agree on specific tasks that need to be completed before the next meeting and assign a lead developer to any new features that need to begin their development cycle.

Minutes

Frequent meetings are not ideal due to one member living off-campus. Instead, we prefer longer, less frequent meetings. With the use of Slack, we can stay relatively up-to-date between meetings.

Meetings will dynamically adapt based on our workloads and tasks. Some weeks we may need two hours of meetings, some weeks we may only need 30 minutes. The minimum per month is 4 hours.

Minutes will be taken by the recorder, and posted to Slack after each meeting. These minutes should include the issues discussed, any votes taken, and what issue were not resolved and need to be addressed at the next meeting.

Meeting minutes will also record who is present as well if anyone was excessively late (with or without a reason).

The minutes will also include any changes to the Team Standards document.

Decision-making process

In cases of disagreements on design choices, it may be helpful to set a concrete process for resolving them. Since we have a group of three members, we will always have a majority in decision making. It is important to value everyone's opinion, though the importance of an opinion also depends on the speaker's knowledge and experience on the topic. Everyone has a right to make an objection in a meeting and propose an alternative solution. They should be heard out, and then a vote held on whether or not to change plans. In the case of only two people being at a meeting, then the third person can be contacted for a decision, or the decision can be pushed back to the next meeting, time permitting.

Attendance

Each team member is allowed to miss 1 meeting per month, provided that there is a legitimate excuse and at least 2 hours forewarning delivered over Slack or by email. There will of course be exceptions for extreme cases. Capstone is about communication. We are all students and all have our struggles, just communicate them to your team.

There are no direct consequences if a pattern of missing meetings is developed, instead it serves as a solid foundation for getting fired. Not warning team members will expedite this process. The same principle applies for being more than 10 minutes late without warning. Direct punishments seem awkward as adults and we would prefer to handle this like a company would: poor performance/attendance leads to firing.

Conduct

In meetings, any major change to the agenda needs to be agreed on by all parties in a vote. If a problem has not been resolved, but it has taken up too much time, then a member can propose a majority vote to move on to the next issue.

If a lead developer is failing to properly handle a feature, then they can be replaced with a majority vote. A lead developer can call for a vote to remove themself if they feel that they don't have sufficient understanding of the problem or think they are otherwise unable to handle this particular feature. Removing a lead developer should only occur after they have received help from other members, and should be a worse case option.

All team issues should be addressed in a meeting with all members present before the matter is brought to the team mentor or Dr. Doerry's attention. If the matter cannot be resolved by the team, only then should outside help be sought.

Tools and Document Standards

Covers all of the tools that will be used, expectations for how they will be used, and related processes.

Introduction

Version control is one of the most important aspects of software development. Having a solid version control process allows for easier debugging, higher quality code, and more effective task management, among other benefits.

To share, maintain, and manage our growing code base, we will use **Git.** The development platform will be **GitHub**, which allows for both public and private repositories for free thanks to student pricing. The **standards** for using Git will be based off a variation of the popular GitFlow. GitFlow is a solid, respected approach to using Git, but it is overkill for our needs as 3 developers and the learning curve will be a tad bit high.

Here we will also define our workflow for **issue tracking** and **communication**, which are closely tied with version control. Formal guidelines will be written using markdown once we create a repository. It is a lot of information up front because there are no assumptions about the team's knowledge of Git and GitHub.

Note: The workflow guidelines are lengthy, but this project is being treated as professional software development. Zach wrote these guidelines for the USGS and they have been proven to work well for small development teams. Although we could 'get away' with less exhaustive guidelines, we would be less prepared for the industry and our quality of work would be lower. Thus, these guidelines should be thoroughly understood and the team will support each other along the way.

Version Control

We will use GitFlow (a 'feature branch workflow') as the basis for our project, simplified for our needs. The workflow description is based on terminal/console 'git' commands. The GUI program 'GitHub Desktop' for Mac OSX and Microsoft Windows allows for visually friendly handling of pull requests, merges, commits, branches, and diffs, but lacks some advanced features (e.g., management of sub-modules). Another option is 'Sourcetree', which provides similar functionality to 'Github Desktop', but covers most of the advanced features that it lacks. Since it seems our development will be done on Linux machines, the top suggestion is the multi-platform GitKraken, which is a modern GUI with nearly-full functionality. As students, we get the pro version of its version control *and* issue tracker that syncs with GitHub for free.

Basics

- The 4 levels of the Git organization
    - **Working directory** (on local computer): work in Atom or similar text editor
    - **Staging area**: include/save changes to the next commit
    - **Local repository**: commit to project history
    - **Remote repository** on github.com: share code with collaborators and backup local branches
- We use **issues** and **milestones** to communicate code enhancements, bugs, priorities, and current progress.
- We have three types of branches:
    - **Master branch**:
        - Any commit on the master branch aims to be deployable. Such commits are usually the result of merging/rebasing with a feature or bugfix branch. Once deployable a unique version number is released. Deployable for us means that commits are tested.
    - **Bugfix branches**:
        - When a bug is discovered on the master branch, a bugfix branch and an issue should be created. The issue should be assigned to the *master* milestone. A bugfix branch should be named after the respective issue. An example of a bugfix branch would be bugfix_16, which represents issue #16.
    - **Feature branches***:*
        - Everything else should be a feature branch, which is where code development is done before it is merged back to master. Each feature branch needs its own milestone. Any bugfixes needed on a feature branch should be directly committed to the feature branch. Feature branches should have descriptive but concise names in upper camel case, such as feature_BetterErrorMessages.
- Testing involves at least that each line of code was executed and did not throw an error or stopped execution unexpectedly. However, writing re-usable unit test cases (GoogleTest for C/C++) is the preferred way to test our code.
- We use semantic versioning using the format MAJOR.MINOR.PATCH. Every commit to the master branch updates the version number. A backwards-incompatible commit increases MAJOR and resets MINOR and PATCH to 0. A backwards-compatible commit adding new functionality increases MINOR and resets PATCH to 0. A backward-compatible commit fixing bugs (etc) increases PATCH.
- Inspecting a repository
    - State of working directory and staging area: git status
    - History of commits: git log --graph --full-history --oneline --decorate
    - List of branches (* indicates the active branch):

    - Local branches: git branch
    - Remote branches: git branch -r
    - List of remote connections: git remote -v
- Finding stuff in a repository
    - Find all commits which have affected a file: git log -- *<part_of_file_name>*
    - Find the SHA of the last commit that affected a file git rev-list -n 1 HEAD -- <file_path>
    - Find all commits which have deleted files and list the deleted files: git log --diff-filter=D --summary
- Error reporting:
    - All *master branch* bugs require you to create an issue in GitHub. Ideally, you provide a unit test which demonstrates the failing code. This unit test serves also as a benchmark to identify the solution of the issue. If it is not possible to write a unit test, please provide a minimal reproducible example. Some resources that may help:
        - How to create a Minimal, Complete, and Verifiable example
        - How to Report Bugs Effectively
        - How to make a great R reproducible example?
        - How to write a reproducible example
    - Assign the issue to the master milestone
    - Create a bugfix branch
    - Close the issue with a reference
    - Create a pull request to the master branch, with appropriate reviewers

Creating Issues

Issues describe suggested new features, a symptom of a bug, a proposed change, and so on. If you create an issue, decide to work on one, or you are assigned to one, then:

- Assign it to a team member and/or yourself, if applicable
- Add an in progress label, if you are currently working on it
- Add a priority label, if it has a low priority or a high priority
- Add a category label, such as 'bug' or 'enhancement'
- Assign it to a milestone
    - Master branch bug: the *master* milestone
    - Feature branch bug/enhancement: the respective milestone

Closing Issues

When the issue is resolved, reference it in the final commit that solves it (which can be done in the title or body).

- Note: Issues will not be closed via reference until the branch is merged to master. If you are resolving an issue on a feature branch, please still use a reference, as it provides beneficial documentation, but you should also manually close the issue afterwards.

Milestones

Milestones map to branches, and hold issues relating to that branch. Whenever you create a feature branch, you should also create a respective milestone.

- Name it the exact same as the branch name, minus the "feature_" prefix
    - For instance, *feature_HapkeModelling* should be named *HapkeModelling*
- Provide an apt description of the branch
- Optionally, provide a deadline

Communication (commit messages, comments on issues, etc.)

Why good messages are important: Erlang: Writing good commit messages:

"Good commit messages serve at least three important purposes:

- To speed up the reviewing process.
- To help us write a good release note.
- To help the future maintainers of Erlang/OTP (it could be you!), say five years into the future, to find out why a particular change was made to the code or why a specific feature was added."

How to write good messages: Who-T: On commit messages:

"A good commit message should answer three questions about a patch:

- Why is it necessary? It may fix a bug, it may add a feature, it may improve performance, reliability, stability, or just be a change for the sake of correctness.
- How does it address the issue? For short obvious patches this part can be omitted, but it should be a high level description of what the approach was.
- What effects does the patch have? (In addition to the obvious ones, this may include benchmarks, side effects, etc.)"

Workflow

1. Set global user options to identify your commits
    i. git config --global user.name <name>
    ii. git config --global user.email <email>
    iii. git config --global core.editor <editor> # e.g., vi, emacs, nano, TextWrangler (Microsoft Windows user refer to First-Time-Git-Setup)
    iv. git config --global merge.conflictstyle diff3 # conflict resolution with three sections: HEAD (code between <<<<<<< and |||||||), feature-branch (code between ======= and >>>>>>>), and (3rd) merged (=last) common ancestor (code between ||||||| and =======)
    v. Activate two-factor authentication for your account on github.com
        - DRS prefers using a TOTP application over text messaging, e.g., Duo Mobile.
        - Command line tools will require a personal access token instead of your regular password.
        - Instead of repeatedly entering the token, you could enable 'git credential caching' with git config --global credential.helper cache (on Linux) or git config --global credential.helper osxkeychain (on macOS).
2. Create a new branch each time you start to develop new functionality or work on improving code.
    i. Get a copy of a remote repository to your local computer:
        - git clone <url>
        - Get a copy of a specific branch:
            - git clone -b bugfix_16 <url>
        - If the repository contains sub-modules:
            - git clone --single-branch --recursive <url> <module>
    ii. Make a new branch and check it out: git checkout -b <branch>
    iii. Push/export local to remote/upstream: git push origin <branch>
3. Create a milestone that describes the purpose of the branch
4. Work on code
    i. Whenever a significant enhancement or issue arises, or when you think one will arise in the future, document it via an issue, and assign that issue to the milestone.
        - A good rule of thumb is that other group members should know what you are working on at any given point in time. After initial functionality has been developed, you should aim to document every significant change that will need to be done before the branch can be merged to master. Close the issue when it has been resolved.
    ii. Stage your changes in a snapshot: git add <file> or git add <directory> or git add -p
    iii. Navigation
        - Between branches: git checkout <branch>

        - Between commits: git checkout <commit> # HEAD points to in a 'detached HEAD' state (i.e., view but do not edit!)
    iv. Remove commits from current (private, i.e., not published on remote repository) state of a branch (i.e., rewriting history)
        - From working directory, staged snapshot, and commit history: git reset --hard HEAD~1
        - From staged snapshot and commit history: git reset --mixed HEAD~1
        - From commit history: git reset --soft HEAD~1
    v. Remove commits from current published branch (by creating a new commit, i.e., it does not rewrite history): git revert HEAD~1
    vi. Restore file from a previous commit: git checkout <deleting_commit>~1 -- <file_path>
    vii. Amending the most recent commit message (see SO for alternative scenarios)
        - Completely rewrite message from scratch: git commit --amend
        - Amend by starting from old message: git commit --amend -c HEAD
    viii. Interruptions to coding: 'stashing' saves uncommitted changes and resets/cleans the working directory, e.g., to switch branches, to pull into a dirty tree, to interrupt the workflow in general. Stashes are handled in the same way as commits by git commands, but they are not linked to a specific branch. Stashes are named <stash@{X}> where X is the number on the stack. For more details see here and here
        - Push a new stash onto stack: git stash (this will only stash files that are already tracked); to stash also untracked (i.e., new files): git stash --include-untracked
        - List stored stashes on stack: git stash list
        - Apply a stored stash: git stash apply will apply <stash@{0}>; apply stash with number X: git stash apply stash@{X}. Git gives merge conflict messages if a stash does not apply cleanly. Apply a stash and stage files as before: git stash apply --index
        - Remove a stash from the stack: git stash drop stash@{X}
        - Apply and remove a stash: git stash pop
        - Show what applying a stash would add/remove to : git diff <branch> stash@{X}
    ix. Resolve merge/rebase conflicts (see, e.g., the section 'How To Resolve Conflicts' of git-merge, here, or here): DRS uses a GUI merge tool (TextWrangler or kdiff3) by issuing git mergetool -t kdiff3. Several other tools are available, see here, here, and here for a comparison and discussion of pros and cons).
5. Commit to your development branch regularly and use explanatory commit messages in order to create a transparent work history (e.g., to help with debugging; to find specific changes at a later time). Each commit is a separate logical unit of change and is therefore composed of related changes.
    i. Check state of staging area: git status
    ii. Commit/save to project history:

        - Commit staged snapshot: git commit -m "<message>" # where contains the commit description on the first line (< 50 characters), a blank line, and a detailed description (include keywords to close/fix/resolve issues, e.g., closes #45 will close issue #45 in the repository)
    iii. Invoke a text editor to compose message: git commit
    iv. *Strongly* follow these guidelines for writing a message
    v. Commit all changes: git commit -am "<message>"
    vi. Commits should be individually compile-able.
6. Share and backup your development commits on the remote repository. Our standard method for publishing local contributions to the github.com repository:
    i. Make sure you are on the development branch: git checkout <branch>
    ii. Make sure the staging area is clean: git status
    iii. Make local commit(s) to the development branch (see previous step)
    iv. In case someone else is working on the same development branch, then import/merge new commits from remote/upstream to local branch: git pull origin <branch>
    v. In case the master branch has changed considerably or contains important updates, then rebase/merge with master
        - git rebase master or git merge master
        - remove/resolve conflicts, mark the resolved files with git add or git rm, and continue with git rebase --continue or git merge --continue
        - and conclude git commit -am "<message>"
    vi. Push/export local project history to remote/upstream: git push origin <branch>
7. Repeat steps 3-5 until the development branch is ready for deployment. Open a pull request and ask for feedback from the team members. It usually does not hurt if someone else than the developer does a test on a feature, since another person may test differently.
8. If necessary, repeat steps 3-5 to complete review of the pull request, e.g. to deal with issues and fix bugs.
9. After a team member has reviewed the development branch, you should deploy the development branch and merge/rebase to the master. Our standard method with two options for deploying a development/feature branch to the master branch on github.com repository (option (i) with rebasing is ideal for small development branches or for simultaneous work on same code section; option (ii) with merging is preferred for large development branches; see following stackoverflow discussions here and here):
    i. Make sure you are on the development branch: git checkout <branch>
    ii. Make sure the staging area is clean: git status
        - Note: submodules will be shown as 'untracked content'. Don't stage and commit submodules. If you want to edit the code of the submodule, do that in its repository and then update the submodule in elsewhere.
    iii. Option (i): Rebase development branch onto the tip of the master branch (given that there are no branches on): git rebase master
    iv. Options (i) and (ii): Integrate with the main code base:

        - git checkout master
        - git pull origin
        - git merge <branch>
    v. Resolve potential conflicts
    vi. Commit and push the merge to remote/upstream with a detailed particularly when using option (ii)
        - git commit -am "<message>"
        - git push origin
    vii. Create an annotated version tag using semantic versioning with a format like v1.0.4
        - Tag the current commit
            - Use git tag -a v1.0.4 -m "<message>" and push the tag with git push origin --tags
            - Alternatively, use the web interface to add a new release against master
        - Tag an old commit retroactively: you should do that so that the tag's date/time corresponds to the commit's date/time by temporarily setting the tag's clock:
        - `git checkout <branch>`

          `git reset --hard <commit SHA1>`

          `GIT_COMMITTER_DATE="$(git show --format=%aD | head -1)" git tag -a v1.0.4 -m "<message>"`

          `git push --tags`

          `git pull`
    viii. Delete the development branch
        - Delete the local branch: git branch -d <branch>
        - Delete the remote branch: git push origin --delete <branch>
        - Remove 'obsolete tracking branches', i.e., branches on local machine that no longer exist on remote/github: git fetch --all --prune

Example

Here is an example of what this looks like in the real world. It shows what the commit messages should be like, how GitHub recognizes commit references, what a pull request should be like, labels, issues, reviewers, milestones, and the code review process. Additionally, there are automated builds, tests, and code-coverage built-in to the repository, which I will also set-up for us.

Links

- https://guides.github.com/introduction/flow/

- https://help.github.com/articles/closing-issues-via-commit-messages/
- http://clubmate.fi/git-dealing-with-branches-merging-and-rebasing/
- https://git-scm.com
    - https://git-scm.com/doc
    - https://git-scm.com/docs
    - https://git-scm.com/book
    - https://git-scm.com/book/en/v2/Git-Branching-Rebasing
- https://www.atlassian.com/git/tutorials
    - https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
    - https://www.atlassian.com/git/tutorials/merging-vs-rebasing/workflow-walkthrough

Style Guide

In development. Depends on language chosen.

A “best practices” guide will not be followed because the learning curve for those is very high and we have a small project. We have a guide for Git, but that can be explained in a few pages, whereas for programming languages the guides are often hundreds of pages. Instead, it is best to highlight a few key points to be consistent on, and just accept that our code will not be the highest quality in terms of best practices due to this not being a full-time software project.

The biggest point here is consistency. Be professional in development. **Code not following the style guide will not be accepted in code reviews.**

Documentation

Comment the code well and write object documentation with doxygen, which is the de facto documentation generation tool for C++ and many other languages.

There are many approaches to the commenting style surrounding the doxygen library. In the case of C++, we will use triple slashes, where the first and last lines are full to 80 characters.

Example:
```
////////////////////////////////////////////////////////////////////////////////
/// @file testFile.c
/// @brief Runs a test to verify doxygen library
////////////////////////////////////////////////////////////////////////////////
```

In the case of Python, we will do the same but with ‘#’.

In-line comments will start with a space and an upper-case letter. Aside from that, just be consistent with spacing and placement.

Indentation

See this discussion which ultimately favors tabs over spaces. We will use tabs.

Naming conventions

For C++, we will follow an already-developed, concise guide. This guide will be the Chaste C++ Naming Conventions, which covers the needed cases quite well without extra fluff. Namely, they do everything with a purpose. *thisIsANormalVariable* but *variables_like_this* are allocated on the heap. You know that *sTestVar* is a static variable due to the s prefix, while *mTestVar* is a member variable.

Other

Without following a large guide, it is impossible to lay out how exactly we want to program. We will just accept that there will be some variations in development style, but an active effort should be made to match the lead developer and base style of the repository. We cannot decide on more guidelines without knowing the language (e.g. should every source file have a header? should we allow an entire class's functionality be defined in the header? etc.).

Word Processing and Presentation

Word Processing will be done over Google Docs in our team Google Drive. Presentations will be done over Google Slides. Any graphics need to have a copy stored on the Google Drive (in a common format) so that other members have access.

Composition and Review

Larger deliverables should be completed before midnight, three days before they should be handed in. The copy editor then has one full day to edit for formatting (Individuals should make sure they check their own work for grammar and typos, that's not the copy editors job). By midnight, two nights before a deliverable is due, a final draft should be made available to all members. All members should read through and comment on any problems with the deliverable. These problems should be handled by the copy editor assigned to that deliverable. The copy editor should handle all problems before the midnight before the deliverable is due, so that time is alloted to print out the deliverable and hand it in.

Team Self Review

At the last meeting of each month we will have time allotted for a discussion about how each team member feels regarding their personal performance. This monthly check-in allows members to receive constructive feedback and input from the other members. These self reviews will be focused on constructive criticism, and should be used to help members be more productive. The overall goal is to be more efficient, and make sure we are each meeting the goals we set for ourselves. When a member identifies an area they struggle in, other members should offer suggestions, and try to help. Review meetings are not the time to air grievances or complain about other members.

During software development we may also choose to transition into a different schedule where each lead-developer is reviewed in a meeting following the completion of a major feature so that he can receive direct and immediate feedback about his latest work and struggles.