# A guide to the HERA Software Community

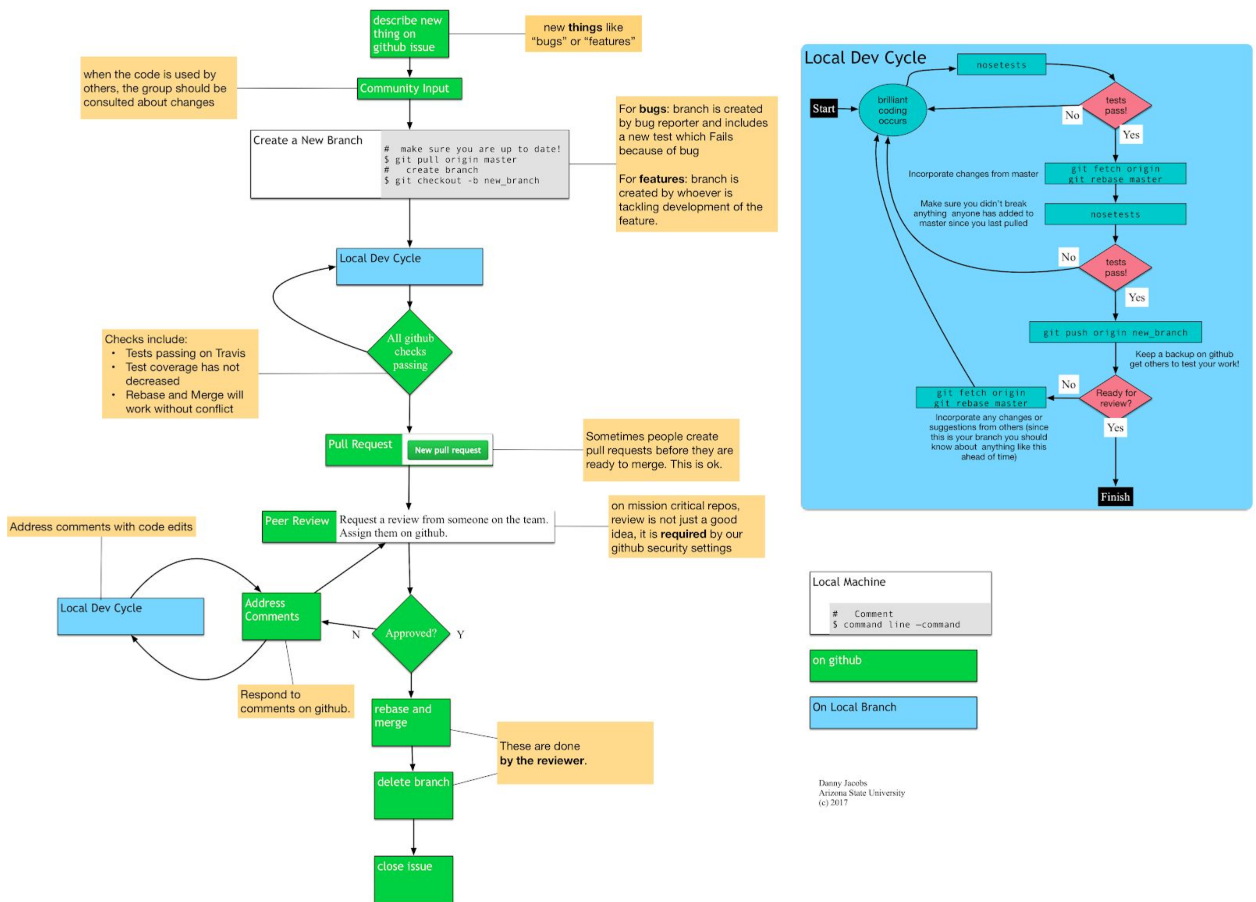
B. Hazelton, & D. Jacobs

DRAFT Version 1:  27 June 2017

Welcome to writing code for HERA!

HERA relies heavily on code at all stages from the correlator to plotting results and everything in between. Errors and inconsistencies in software development can have a consequences ranging from lost time to lost data. As a community we have arrived at a set of practices that seem to work well for our group. This document will get you up to speed.

Danny's Software Dev Cycle




Note: in the final document this should be like a fold-out double page or something.

- Criteria for peer review

- Documentation (doc strings, sphinx, tutorials for new features)
- Test coverage
  - Test coverage measures the percent of relevant (i.e. functional, not comments or continuation) lines of code that are covered by unit tests.
  - Test coverage is best measured by setting up Travis and Coveralls on the repository. Travis is a continuous integration (CI) service that downloads the repo code, installs it and its dependencies, runs the tests and reports the results. It downloads and runs every time there is a push to the repo and you can see the outcome for every push on every branch. In addition it tests whether pull requests would pass after being merged into master and can report those outcomes on the github pull request conversation.
  - Coveralls is effectively an add-on to Travis that reports code coverage for all Travis runs. Coveralls provides a nice interface that shows the code for each file with lines that are covered by tests highlighted in green and lines that are missed by tests highlighted in red. These file level details be seen by branch and even by commit. Coveralls can also report coverage increases and decreases on pull requests and can be configured to "fail" the coverage test if the coverage drops too much.
  - The goal of test coverage should be 100%, but above 95% is a reasonable requirement for project-level code. Coveralls should also be configured to not include the test code in the coverage count (which artificially inflates the coverage percentage by adding many lines of test code that always gets run). This is obvious if you look at the files with coverage details on Coveralls -- the test files should not be listed.
- Pep8
  - Pep8 is a python style guide that specifies a range of things from formatting to good code practice with the goal of making code more readable and reliable ([https://www.python.org/dev/peps/pep-0008/](https://www.python.org/dev/peps/pep-0008/)).
  - We do make an exception for E501, the Pep8 code for lines that are longer than 80 characters. While we feel that lines should generally not exceed 80 characters by much, we do not feel that enforcing a hard cut at 80 characters makes code more readable, which is really the goal. That said, having to scroll to the right in your editor to read a line of code probably means the line is too long and should be broken up.
  - There are several tools to help you ensure that your code conforms to pep8. Most of these tools rely on the pycodestyle package which can be run as a standalone tool to identify lines that do not conform to Pep8 standards.
  - Perhaps the most useful use of pycodestyle is by editors such as Atom (also Sublime and maybe Spyder) that use linters to inform you as you write your code of Pep8 infractions, making it easier to keep to the style as you work. You can also set Atom to ignore certain codes, like E501.
  - There are also command line tools like `autopep8` and `flake8` that may be useful, but you should be aware that automatic formatters cannot address all Pep8

        issues (e.g. bare except statements), so after running an automated formatter, you should also check that there are no remaining issues using a tool like pycodestyle that just reports the problems.
    - Interfaces, data standards, and coding practice
- Documentation
    - Standards: docstrings, tutorials, functional test memos
        - Doctests for tutorials and examples in docstrings. Can be run via Travis as well.
    - Continuous (sphinx)
    - Automating build of docs from the code
- Testing
    - Unit testing vs functional testing
        - Everyone who writes software tests their code at some level to make sure that it behaves as expected. We draw a distinction between two types of tests, primarily defined by whether they can be run automatically without human oversight or whether they really require human intervention or interpretation (because they use several disconnected code bases or require value judgements, etc). We call the former type “unit tests” and the later type “functional tests”.
        - Unit tests can be run automatically by a computer and are constructed by defining exactly what outputs (including errors or warnings) the code is expected to generate under specific circumstances and asserting that the code does indeed return those outputs.
        - In many ways, unit tests represent the lowest common denominator of testing -- they are often very simple minded, testing only known behavior and errors and they sometimes don’t feel like they are worth the effort that it takes to write them. In practice, however, they are **extremely** useful tools for catching bugs quickly in a controlled environment that can save software developers a great deal of time hunting problems through a complex software stack. They also guarantee code stability when changes are made because they help the developer find and fix any downstream consequences of their changes (which may require updating the tests as well as the code).
        - Unit testing is necessary but not sufficient! There are many complex and subtle bugs that do not get caught by unit testing. That is what functional testing is for, but once a new bug is identified (either via functional testing or a user bug report), a new unit test should be developed that fails because of that bug. Then when the bug is fixed, that unit test will help prevent similar bugs from coming up again in the future.
        - functional testing covers a wide range of test types and should really be tuned to the context of the code in question. The kinds of things it can cover include:
            - round-trip testing with other packages to be sure that the inputs and outputs are as expected
            - using simulated data to test data processing software to see if the outputs behave as expected

            - human interpreted plots that show that complex algorithms are generating the expected behavior
        - It is good practice to document functional tests, particularly complex ones, in memos, tutorials or other documentation in the project repository so that they can be re-visited or re-run as needed after significant code development work.
    - Unit testing
        - We use the Nose unit testing framework for python (http://nose.readthedocs.io/en/latest/). Nose supports the classic python unittest class framework as well as other test classes or standalone test functions.
        - Nose will automatically find and run tests in any package, folder or file if the package, folder or file is named test_* and if the function or or class is called test_*.
        - If you choose to write tests in unittest subclasses, you can use the assert statements that come with those classes, otherwise nose provides many assert statements (assert_true, assert_equal, assert_raises…) via nose.tools.
        - To run the unit tests on a repo, just type `nosetests` in the top level repo directory.
        - You can also limit which test files get run by passing the relative path to the file or direct nosetests to exit on the first error or failure using the option `-x`.
    - functional testing
    - Automated testing (travis)
    - Test coverage (coveralls)
    - Examples: finding a bug, testing errors and warnings, ex wenyangs heracal test memo
- Coding practice
    - Inheritance (overloading is bad)
    - Variable naming preferences:
        - No single character variable names:
            - bad - for t in times:
            - Good- for time in times:
        - Units in variable names (where relevant)
            - Bad: baseline  =  2
            - Good: baseline_wl = 2
    - Installable python repo structure, folder conventions.
    - Scripts thin wrappers around modules (this makes it possible to cover everything with tests).
    - Git rebasing
- Data standards
    - Metadata lives with data
    - Code versions are part of metadata
    - Data doesn't live in code

Systems (how they are set up)

Git & GitHub

Everyone writing code for HERA is a member of github.com/HERA-Team's "Dev team"

For sw powering critical systems or used by a community, permissions to commit to master are limited. All normal dev must be reviewed as a pull request from a branch.

Github repos should have their merges restricted to rebases (under settings/merge button only 'Allow rebase merging' should be checked)

- Travis
  - How to set it up
  - Example travis.yml file
  - Adding badges to your readme
- Coveralls
  - How to set it up
  - Adding badges to your readme
- Slack
  - Channel integrations (which channel, how to choose)
  - Discussions continue where they started (don't respond to a PR conversation in Slack)

HERA Repos

- capo:
  - deprecated, only continue to be used for PAPER analysis
- hera_sandbox:
  - play area for developing trial code
  - NOT to be used for HERA operations or published results
  - not installable and contains only scripts
  - many old things migrated here from capo
  - each person has their own folder
- pyuvdata:
  - Standard library for reading/writing and converting file formats for:
    - visibility data
    - calibration solutions
    - beam models
  - Intended to have a wide audience (i.e. not just HERA) so must support a wider set of radio astronomy standards.
- uvtools (modules from capo)
- omnical
- Real Time Processor (RTP)
- Librarian
- Monitor and Control (hera_mc)
- Quality Metrics (hera_qm)
- Hera Calibration (hera_cal)

Appendix A: Git rebasing

First, make sure you're coordinating with any other user that is contributing to your branch. If any other user has your branch checked out, they will need to delete their local copy of the branch and check it back out from the remote after you finish rebasing and push that rebase. If they have local changes they want to be included in the rebase, those changes need to be pushed to the remote and you need to pull them into your branch before you rebase.

The basic process for git rebase is that it rewinds all the changes on your branch since it was last in sync with master, pulls in the changes to master, and then tries to reapply the changes on your branch one at a time. After making sure that you are up to date on both branches, switch to your branch and type `git rebase master`.

You can occasionally get a merge conflict in the process, if your branch and the master has edits to the same part of the code. If there is a conflict, git pauses the rebase and asks you to fix it. At that point you go into the indicated file(s) and decide which changes to accept. It's important to realize that this might be partly through the process of reapplying changes so it might not contain the most recent code. After you fix the problem, you should git add the file(s) but DO NOT COMMIT! Then type `git rebase --continue` to continue the process.

Once you have finished rebasing DO NOT PULL. Instead you need to do a force push (`git push -f`) because you you are rewriting history.

Ask for help if you have any difficulties. If you get stuck midway through the rebase and don't know what to do, you can always bail out with `git rebase --abort`.