Contribution from Wolfgang Dobler

to Issue 2025/1

May 21, 2025, 2024, Revision: 1.2

Contents

1 The problem(s)

The currently dominant development model of the Pencil Code is

- *unstructured:* Commits cannot be grouped predictably, because while you try pushing them, others could push some commits of their own;
- *opaque:* if commits are not grouped, it is hard to see what is going on;
- *hard to test:*
  - Continuous-integration tests (= CI tests, which we sometimes call Pencil tests or autotests) have hard time constraints and thus cannot be thorough;
- *fragile:* as a consequence of being hard to test;
- *obscuring responsibility:* If I made two commits, and A has pushed 3 commits in between (see *unstructured* above), would I be testing my changes or theirs?

2 Advantages of branch-oriented development

- Much deeper testing
  - You can take your time to run all tests before merging your branch back into *main* (= *master*). This allows us to have really thorough tests.
  - Independence from CI. CI is nice, but it is dangerous to rely on it (and it limits the amount of testing)
- Clearer identification of responsibility
- Isolates unrelated changes from each other until they really need to get dealt with
- Friendly competition (survival of the most popular branch)
- You can get new features running and ask interested others to test before you finalize them (instead of: everybody immediately has to live with your new feature, although it is not thoroughly tested yet – you had to push it onto *main* before asking the interested others).
- Reduces stress level, because you can properly work on a feature without interfering with others and later do all the tests and push.

3 Note

- We have used branches before (e.g. the GPU branch)

4 Feedback and criticism

Here is the gist of some responses I got to an earlier version of the proposal below.

- "Risk of fragmentation"
  - But where is the problem if all branches are pushed to the server?

    - If you need functionality from another branch on your own feature branch, you can (a) merge that branch, or (b) cherry-pick the relevant changes from there.
  - *The important thing is to have all branches pushed to the server.*
    - Currently: If you are not willing (yet) to push, your changes are only local, which is much worse.

- "For short-lived branches which are not shared with anyone else, I would recommend rebasing, as it keeps the history cleaner (this helps later on if one is bisecting to find when a bug was introduced to the master branch)."
  - I am open to that, though I personally find history with (even medium-sized) topic branches cleaner; and `git bisect` works across branches just as well.

- "I fear that if we start using more branching this will result in some branches never being merged back into main. This will eventually lead to several versions of the Pencil Code."
  - There will always be some branches not (yet) merged. If they are relevant, they will eventually get back into *main*, be it that they get merged or cherry-picked. (And if they are not, who cares?) Should there really appear two branches of the code that continue to be popular among users over a long time, we should strive to merge them.

- "...other codes in which multiple groups maintained branches of code which were entirely incompatible with the main branch or each other."
  - Rule 6. below strongly encourages to not let branches diverge too far from *main*. Being able to decide for yourself when exactly you reconcile your changes with *main* is an enormous benefit.

5 Proposal

Here is a set of rules that I would like to see implemented:

1. The main person responsible for a commit is its author.

2. The maintainers are important, as often only they know how a sample really works. However, it is not their responsibility to react to commits breaking an auto-test sample.

3. Ideally, any commit that ends up on *main* (the main branch of the code) should not break any of the essential auto-test samples.

   If it turns out that a commit violates this rule, the broken tests need to be fixed swiftly. Either by the commit author committing a fix, or by others reverting the commit in question.

   Any commit that breaks an auto-test may be reverted by anybody without prior notice in order to fix that test.

4. Keep individual commits focused on one logical unit (i.e. keep them small and do not conflate loosely related topics). This is crucial for identifying and reverting problematic commits.

5. For a sequence of commits that belong logically together, it is not very feasible, and also not necessary, to run the test suite for each of them.

   Instead, develop the feature on a branch. When you think the feature is ready, merge *main* into your feature branch and run the tests. If they all run successfully, merge your branch back into *main*.

   Never merge a branch into main *without having verified that the auto-tests work.*

6. Try to keep the lifetime of branches short, even though this will not always be possible. If your branch has to live longer, you should periodically merge *main* into it.

7. *Never quietly adjust reference results in an auto-test sample directory to fit your latest data.*

   If reference data need to be touched up, this must be agreed upon with the maintainer(s) of that test.

8. When making changes that will affect auto tests:

   (a) Discuss this with everybody before you commit the change.

   (b) Isolate auto-tests against the change. E.g. if you are to change the default value of `lmassdiff_fix` from `.false.` to `.true.`, make sure to explicitly set `lmassdiff_fix = .false.` in all auto-test samples that would otherwise be affected by the change.

9. Never rely on the automatic tests run after a commit on the GitHub server, or the bisection that is done. It is the committer's responsibility to run all officially accepted auto-tests.

10. Never rely on automatic emails to reach a maintainer of an auto-test (or in fact anybody else). If you need to discuss an auto-test, contact the maintainer directly and only assume that you reached them once they reply.