A Workflow for Increasing the Quality of Scientific Software
(the slides are available under “Presentation Materials” in the above URL)
Date: April 7, 2021
Presented by: Tomislav Maric (TU Darmstadt)

Q. Reading ahead in your slides, it looks like you are having your users create a fork and submit a merge request onto the main repo. We have tried running in this mode but have run into an issue, the other developers on the main repo cannot see the CI pipeline status of the merge request unless they have read permission to the user’s repo. I tried for a while to fiddle with gitlab options to make this work but couldn’t find a solution. Did you have similar issues and if so, how did you solve them?

A. Yes, we are requesting access to CI pipelines (members of the project can be given access on GitLab) if we are working with merge requests, for repositories where things are even more simple, and we are working directly on branches, this is also ensured. Since the workflow is aimed at University research groups - there should be no problem with this. Access to the CI pipeline and project membership for projects from external collaborators can be requested as a prerequisite for the contribution to the project.

Q. How much of their time do the members of each team actually devote on code development for the project? How long are periods that things “don’t move ahead”? What other responsibilities does a PhD student usually undertake in your group?

A. Most of the time is spent on development. We sometimes need months before things “move ahead”, and this time is spent in the cycle of analyzing results in Jupyter notebooks, discussing them and implementing alternative algorithms. Since we don’t know what will work and what won’t work, we rely on dynamic and static polymorphism in C++ to combine sub-algorithms, which requires effort. The implementation and debugging takes most of the time also because the methods are “unstructured” (unstructured domain discretization) which complicates the data structures and algorithms used to manipulate geometrical approximations of fluid interfaces, and our task is to keep everything simple, accurate and scalable, that takes a lot of work.

Q. As a percentage %, how much time do the members of each team typically devote to requirements, software architecture, software design, and software testing for a project?

A. In numbers, my subjective estimate (speaking for our team): postdoc ~60%, PhD student around ~30% at the beginning and -60% towards the end, masters students ~20%-30%, bachelor students ~5-10%. Maybe this is specific to our numerical methods, but without enabling the selection of sub-algorithms (at runtime if HPC allows, at compile-time if not), without automatic testing of parameter variations, we would spend 90% of our time manually running all the tests each time we implement a new algorithm and examining them. There are

just too many options - take for example the sub-algorithm groups just for the interface motion by the geometrical Volume-of-Fluid method: interface positioning, interface normal estimation, interface normal optimization (2nd order), temporal integration of displacements, cell center to cell corner-point interpolation / extrapolation, flux volume triangulation, phase-specific volume intersection, and error correction. Imagine we have at least two concrete sub-algorithms in each group, and all of them can be combined with each other to find the best method over a set of verification cases, then imagine running all those combinations of algorithms manually. The time we invest we gain back from automated tests, and also from the ability given to us by software design / architecture, to extend the code very quickly, instead of spending months figuring out how to insert a new functionality into a monolithic and rigid code-base.

Q. Going back to the first item on the list, I wonder if you can comment on the Kanban board - your experience with using it, getting others to use it, to gather around it on a regular basis. Paperwork and status meetings are often low on people's list of priorities… and yet, it seems vital to all other efforts.

A. We are very happy with Progress Tracking Cards: we write down a larger task that we are working on next, separate it into sub-tasks, each of us commits to specific sub-tasks, and we hack away at them together as a team. We discuss in Slack and meet on Zoom as soon as something needs to be talked about in more detail. This worked much, much better, than designing the project as an Agile project with Epics, Stories, Tasks, doing Sprints, or even using Kanban boards with each of us working separately on separate issues. There is a psychological aspect of PTCs I think that makes them work: it is more clear that we are working together as a team on the same goal (task), we all psychologically commit to sub-tasks by defining how exactly they will be done (not an empty "yes I'll do it"), and the sub-tasks often relate to each other, so we need to engage each other more, which strengthens the team.

Q. How do you get around the problem where one student or postdoc ends up doing everything, and ends up destroying their chance for career advancement in academia?

A. This is a very difficult question, and it hits home. I'm myself in a similar situation, because focusing on numerical method development and taking the time for software design and testing reduces the number of publications. In fact, I recently got a very negative feedback for my meager publication frequency and was rejected after applying for a research grant. Still, I would rather do research properly and fail to advance in an academic career, instead of "playing computational science": "scientific" code working for a small sub-set of tests that get published while other failing tests are ignored, choosing physical parameters that hide the issues, numerical methods that whose inputs are tuned from one case to the next differently in order to publish a paper, not being able to develop the methods further because the code is completely not maintainable, and similar computational science anti-patterns.

Q. Which system do you use for Kanban boards? Trello? Simple post-it notes?

A. .GitLab issue boards.

Q. You propose a workflow for improving the quality of CSE software but do not define what you mean by [Quality]. How does this relate to software quality, i.e. non-functional requirements, as it is broadly understood by the field of software engineering?

A. We define quality of scientific software from the functional perspective, because already this is severely lacking in our research community: source code or research data are not available, research data is not associated to a specific version of the software used to produce it, test coverage is extremely poor and hides issues. From the non-functional perspective, the software should be sustainably developed: this requires at the minimum integration of versions. Already the integration of versions becomes a serious problem if the software is not modular: if the numerical algorithm is not extracted and exchangeable, it will be hacked, just for the next publication. The software should be written in a way that allows extension without modification as much as possible, and clean code is the basis for this. With C++/OpenFOAM, it is quite clear what needs to be done, since the available design patterns prescribe how to extend it while maintaining a high level of software quality. It took time to reverse-engineer those, but that's clear now. The modularity needs to be addressed by the project Maintainers when integrating changes from other members in the team. Reducing number of function arguments, argument commutativity, small function bodies, abstract types, patterns, etc. Beyond inciting researchers to invest time and learn about design patterns and clean code I don't know how to provide a general guideline at this point. Maybe for C++, already understanding dynamic polymorphism, Build Pattern with runtime type selection and the Strategy Pattern, gives enough "Lego" blocks for significantly increasing modularity thus simplifying automatic testing and finally Continuous Integration.

Q. Are maintainability and extensibility not core to achieving software sustainability, i.e. longevity or technical sustainability?

A. Yes, definitely, see the answer above.

Q. You mentioned modularity as a criterion, what design criteria do you use to address coupling and cohesion

A. We don't explicitly think in terms of coupling and cohesion - we think about building blocks of a numerical method. Take for example unstructured Front Tracking for two-phase flows. It needs to *reconstruct* the fluid interface as a surface mesh, *locate* Eulerian mesh cells that store surface mesh points, *interpolate* the velocity from the Eulerian mesh onto the surface mesh, *integrate* point displacements, and so on. The italic words are our abstract classes, their member functions take as arguments global variables modified by the Partial Differential Equation solver application, like *interpolation.interpolate(eulerianVelocity, frontVelocity)*. We ensure that the Build Pattern is used that enables selection of concrete classes at runtime based on configuration file entries. If any abstract class contains complex sub-algorithms, we

apply the Strategy Pattern for the sub-algorithms, and make them also runtime selectable based on configuration file entries. So basically, coupling and cohesion will spring out naturally from the structure of the numerical method and global variables used in the Computational Fluid Dynamics “main” function. When we notice we messed something up, we refactor.

Q. On slide 8 you mentioned the principle of Single Responsibility. Is there a reason that you have not incorporated the other four principles from S.O.L.I.D, i.e. Open-closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and the Dependency Inversion Principle

A. We incorporate whatever we can, but we keep things as simple as possible, since we are not educated as software engineers, and can’t avoid the publish-or-perish pressure, we found that already using abstract types (above answer) with SRP and the Strategy Pattern and separating code into layers (Separating Concerns) significantly helps to remove git merge conflicts, and it streamlines contributions to the code.

Q. For my personal experiences, usually most users tend to use the “development” branch and then the “main” branch becomes little interested. Do you have any comments or suggestions?

A. When we reach the milestone and do the cross-linking of source code, publication / tech report and research data, we merge with the “main” before creating the git-tag. This way, we do it every couple of months. The cross-linking step makes us look at everything once more before submitting to peer-review and clean up whatever we can.

Q. Related to above - there’s a difference between ‘Paper accepted’ versions and main-branch released versions of the CSE tool in this workflow, this differs from general practice for users of scientific software where at the very least they are asked to quote a version in their paper. Have there been cases where published results (based on a feature branch) have had to be retracted or otherwise significantly differed from the eventual main-branch version that was released for general re-use ?

A. We use “git tags” to identify versions that we then cite in publications. When we submit to peer-review the git tag might look like 2021-05-15-Submission, then if the paper goes into peer-review, JCOMP-30212-R1, JCOMP-30212-R2, then at some point finally JCOMP-30212-Accepted. We edit git tag descriptions and fill in metadata that link the git tag with datasets and the publication (arXiv ID, DOI for *-Accepted). This way, we’re not submitting to peer-review from the feature branch, but from a merge first into “development”, then to “main”, then we generate the git tag, and we create new git tags during the review process.

Q. Are you also writing unit tests in OpenFOAM? If yes, which framework are you using?

A. Google Test **but:** as mentioned in the talk, we don’t do mock-ups of OpenFOAM classes or mock-ups of input data. A unit-test for us is a test that ensures a smaller sub-algorithm, a

building block of a large numerical method, works as expected. We test this algorithm on input data that's as close as it can be to real-world input.

Q. I find that OpenFOAM makes modern development a bit difficult due to some special approaches, such as using WMake. What good practices would you advise me to follow to be able to apply modern techniques in developing for OpenFOAM?

A. Well, we've compiled a couple of projects with CMake, here is an example project with CMakeLists.txt for OpenFOAM. This took some effort but it is doable. WMake complicates the CI, but if the recipe from the slides is applied for artifacts, it works. I believe in the end, as we release everything we did, we'll have both Wmake and CMake enabled in projects.

Q. What code-related skills would you wish that M.Sc. CSE students acquire? You cannot answer "version control" or "testing" :-)

A. Dynamic polymorphism, object-oriented design patterns. (follow-up: opened an issue to add a few of these to our programming course next year!)

Q. You mentioned BSS's Progress Tracking Cards (https://bssw-psip.github.io/ptc-catalog/catalog ?) - are there any public projects from your colleagues which show how the kanban board have been used with these PTCs ?

A. Email me, I'll be happy to add you as a guest to such a project: issue tracking is not visible to everyone, as we can't really share our ideas with the world without even having published a preprint at least.

Q. At least from my personal experience with CFD software such as OpenFOAM, the top-down approach to code testing involves testing against exact solutions of the Euler or Navier-Stokes etc. equations that are usually made up by introducing an unphysical forcing function on the RHS. Is this the type of test you are talking about? We still used to find that especially for 3D turbulent flows these tests were not always catching errors, especially because they tend to be steady state solutions. More generally, how do you choose those tests with a right balance for testing cost and coverage?

A. The method of manufactured solutions is useful for PDE verification, but we run test cases that are "physical", in the sense that we compare with others and with experiments. In these cases, we are not that far, that we automate fail / pass evaluations for such tests. In this sense, the Continuous Integration is not really continuous, as we meet as researchers and discuss the results in Jupyter notebooks and then check primary (simulation) data as well. We use coarse tests so far on workstations to develop the methods up to the point when we can move to the cluster. Everything the CI pipeline does automatically on a workstation, we can re-use on the cluster: submit parameter variations with production input and monitor results using Jupyter notebooks. This manual step we didn't avoid yet, since, as mentioned, we can't catch all errors

by a relatively coarse input.

Q. Also if comparing against a “known solution” from a previous “blessed” version of the software, how important is it to get “close” to binary reproducibility and to the degree that you do not ask for that in a strong manner how much do you allow for δE type of errors on integrated quantities (or do you expect those to be identical to the last decimal?) or pointwise ones? If allowing for an error bound which norm are you using?

A. We are definitely not expecting to reach any kind of point wise reproducibility in terms of floating-point accuracy. Even for the CI tests, we examine notebooks, look at the absolute errors and rates of convergence, and literally see if we have improved with respect to the last version - compared to the last version’s notebooks. We haven’t tried to quantify this improvement and fully automate CI this way for physics in the “fail until everything is better than last time” sense. Apart from tests where a failure is perfectly clear and quantifiable easily (e.g., diverging interpolation method, numerical bound [0,1] exceeded), for validation testing (comparing with physics), we examine everything ourselves, it is a part of the research cycle.

Q. With everything you just listed, you basically need to be a full-fledged software engineer to write scientific code. So how do we keep these people when the FAANGs of the world will pay them 3X what we will?

A. I wouldn’t say one needs to be a full-fledged software engineer, because even if the workflow is partially adopted, the quality of scientific output increases. Already with version control and periodic integration and cross-linking, at least the functional quality will increase significantly. It becomes possible to find secondary and primary data behind a publication, as well as the source code used in the publication, and vice-versa. Version control, integration and cross-linking cause a small overhead with a huge benefit. Tests need to be written at some point to publish a paper, how else to get the research results? Why not write them immediately then, before starting to program numerical algorithms? That’s practically the nutshell of TDD, and it does force one to think about the API from the user’s perspective, which increases non-functional quality for egoistic reasons - we don’t like to use bad code even if it’s our own. So even if Continuous Integration is not done, care is not taken for secondary data formatting, and the parameter studies are not organized for future comparison, there is still a benefit from a minimal workflow.

Regarding the salaries, from my personal perspective, non-commercial research allows for freedom and creativity not offered elsewhere, and researchers that are drawn to science and not income or social status care deeply about reproducibility, robustness, accuracy and applicability of their research. If they are taught software engineering practices early in their research careers, or even during their studies, they will use the practices naturally from the start. Those people one should keep, and in my personal opinion, those people will ignore 3x larger salaries and stick to research as long as they possibly can. As for the rest, for the time they spend in research institutions / Universities, their knowledge of software engineering will improve the

quality of scientific output of the institution / group, and improve their C.V.s for the industry, so everyone benefits.

Q. How does exploration fit with research as TDD ?

For two-phase flow simulation we have canonical verification and validation test cases that are known for decades, since the beginning of method development in the 1960s at Los Alamos mostly, and our goal is to build better numerical methods that improve standardized errors in these tests. The exploration is done in the sense of adding new algorithms in the numerical libraries, with known tests.

However, TDD works well with exploration as well. If it is entirely unclear if the idea will work and the idea does not fit into an existing model hierarchy structure, we literally first hack it into the solver application and run the tests with this new hacked explorative method. If the initial results show promise, then we extract the hacked code into the library and clean it up in the first refactoring step, before improving it further. We don't invest time into clean code for every idea from the start, that's what TDD really helps us with, and that's one of the reasons we test first. However, if the new idea does belong to the existing hierarchy, we can add it very quickly in a clean way, because it usually simply means implementing an abstract interface, or specializing a C++ template.