Useful Practices for Software Engineering on Medium-sized Distributed Scientific Projects

Dan Gunter (dkgunter@lbl.gov), Keith Beattie (ksbeattie@lbl.gov)
Lawrence Berkeley National Laboratory

Introduction & Background

Modern science depends heavily on computing and computer software. The correctness, efficiency, and more generally the *quality* of the software is instrumental to scientific advancement. Software engineering -- "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [*IEEE Standard Glossary of Software Engineering Terminology*, IEEE std 610.12-1990, 1990] -- is the major tool available to create and maintain software quality. Advances in software engineering such as "agile" processes and the "devops" revolution, have been important factors in the creation of scientific software.

In our work, we are often in the position of leading, or helping to lead, the software engineering efforts of medium-sized, distributed, multi-disciplinary scientific project teams. By *medium-sized*, we mean from about 10 to 50 people; by *distributed*, we mean that the team is comprised of people from multiple institutions that are geographically separated and do not have a common management structure -- typically a few universities and one or more national laboratories; and by *multi-disciplinary*, we mean the team is usually composed of people who come from scientific or engineering backgrounds other than Computer Science or Software Engineering, with limited experience writing software for use outside their own or team's projects. We have often tried to apply software engineering and project management approaches from industry in this milieu, but always end up frustrated with some of the assumptions about centralized authority, dedicated software engineering effort, and incentive.

A good example of such a project team is the Institute for the Design of Advanced Energy Systems (IDAES) [1], which was formed in 2016 to develop new advanced Process Systems Engineering (PSE) capabilities. Funded by the DOE Office of Fossil Energy, and led by the National Energy Technology Laboratory (NETL), the project's mission is to improve the efficiency and reliability of the existing fleet of coal-fired power plants while accelerating the development of a broad range of advanced fossil energy systems. The IDAES team spans three national laboratories (NETL, SNL, and LBNL) and three Universities (Carnegie-Mellon University, West Virginia University, and Notre Dame University). Most of the team is in two locations -- CMU and NETL -- in Pittsburgh, PA. However, there are half a dozen participants at each of LBNL (Berkeley, CA) and Sandia (Albuquerque, NM), as well as professors and several

graduate students at WVU and Notre Dame. From a software engineering perspective, the goal is to build a software package that is capable of simulating important aspects of power plants and power grids. Many of the ideas for why and how to build the IDAES software came from experience on an earlier project, the Carbon Capture Simulation Initiative (CCSI), that used commercial tools to perform the process simulations. Though the target users of this capability are power plant operators and power grid designers, the main user base is currently the developers and graduate students themselves. Application of the tools to external customer problems is currently performed at the level of the internal team themselves building the models and iterating with the customer.

Approach

While the IDAES project runs functionally as a unit, there are sub-hierarchies of control in each national laboratory and university that set local priorities within the framework of the project deliverables. And although most of the effort is being spent on new software to perform chemical process engineering, there is also significant effort in adding related capabilities to an existing software package, Pyomo [2] [3], that pre-dates IDAES and continues to be developed at Sandia National Laboratories. This size and variation in scope makes project coordination a challenge, but also is not unusual in our projects.

Although the specific approaches to increasing overall team productivity through software engineering vary across projects, three elements of what we are doing for IDAES are, we believe, generally applicable to projects with this kind of mix of institutions, disciplines and scale: weekly whole-team developer meetings, incrementally better automation, and “soapboxing” (including software engineering in official goals & deliverables).

Many styles of meetings have been discussed in recent years with respect to software development teams, most commonly the daily “standup” meetings combined with some semi-weekly longer “retrospective” meetings. Neither of these cadences fit what we believe is a unique constraint of our scientific environment: that there is no common authority structure (due to the multi-institutional nature of our collaborations) with contributors split across timezones, (sometimes continents) and multiple unrelated projects that have no inherent interest in accommodating each other. As a result, people generally have very limited slots in their calendar that they can guarantee are “free” on a regular basis. However, one weekly hour-long meeting is generally possible, and serves multiple purposes: (a) let people “context-switch” back to what they promised a mere week ago to have done, (b) provide an open forum for cross-cutting issues or questions that don’t get easily addressed in subgroup meetings (because, of course these also exist), (c) provide some opportunity to disseminate practices and educate new team members, (d) build camaraderie through regular (virtual) contact. For all these purposes, it is important that this meeting is open to the entire developer team – however daunting that must seem at first. You can think of this meeting as a little bit like going to a weekly religious service or tight-knit social group (church, temple, book group, bicycling club, etc.) where the enthusiasm for the event may vary week to week, but the overall experience is as much about the custom and connections as the content.

Specifically what occurs during these meetings will probably depend on your development practices and tools. But in general, we have found that there should be two main activities: an open agenda of issues that people can edit before the meeting; and a standard task that focuses attention on the active development across the project. For many projects, a modified Kanban [4] “project board” approach provides an excellent focus because it provides an easily summarized list of things done, in-progress, and abandoned. Whoever is leading the meeting – and of course there should be just one person who leads it, since anything else with 20+ people on the line would be chaos – can in the absence of any other topics simply walk through all issues, categorize them against release or other timelines, etc. Almost always, this simple practice will bring up interactions between the different pieces of ongoing work that would have otherwise been discovered much later, if at all.

Between meetings, the best friend of overall team productivity are automated practices, and in particular automated tests. Although very few people disagree with this in principle, the truth is that getting a large suite of automated tests to work, and keeping them working in the face of constantly changing software, personnel, dependencies, etc., is non-trivial. Of course, most of the tests must be written by people who understand the mathematics and science of the code being tested (or, in the case of infrastructure code, the computer science principles of the code). However, we must face the reality that this is a significant request of someone’s time, and is competing with the next publication or result, which they may be encouraged to prioritize. Our position is that the right approach to take to this problem is incremental: help people put in a few tests, but don’t require that they pass; then add a requirement that the tests pass before the code is merged, but don’t worry about code coverage or style; then start informally checking code coverage, and style, but don’t enforce anything; then start enforcing low levels of code coverage and very basic style rules; and finally – well, honestly we’ve never gotten farther than that. But by doing this all in baby steps, and talking about it weekly in the developer meetings, you are building up a culture of testing that will lead to eventually having a robust software environment that is broadly supported by the team.

All told, these meetings serve to address the challenges we face developing software with teams composed of not primarily computing or software engineering people working together without a common authority. The regular meeting baby step approach both educates and builds consensus simultaneously on the best procedures, practices, and tools to adopt. Ideally this approach is driven by demonstrated effectiveness from those with the most software development experience.

The final element that we think is useful is what we call “soapboxing”, in the sense of standing up on a soapbox and shouting out to the world, by putting key elements of software engineering and development practices where project management and funders can see them. Probably the least important place for this is grant proposal text, since these are read at most once every few years, and this is definitely a case where repetition is needed. One of the key places to try to insert some discussion of software practices (and by extension, developer productivity) is in project meetings. If your PIs and other project leads need any convincing that this is an

important topic, an excellent way to do so is to show them the team engaging in a spirited discussion in a public or semi-public forum. Progress reports and general-audience publications about the project are also opportunities to describe software engineering practices.

Conclusion

We have spoken largely of social challenges, but of course there are numerous technical challenges that scientific software presents to developer productivity, which must also be addressed. We believe that conquering these technical challenges goes hand in hand with reducing the friction of the software engineering environment so that developers in medium-sized teams don’t spend all their time working in silos and unintentionally stepping on each other’s toes and, perhaps most importantly, feel like they are working as a single team no matter how different their technical backgrounds.

References

[1] "IDAES Project Web Site," 2018. [Online]. Available: http://idaes.org/. [Accessed 28 June 2020].

[2] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson and J. D. Siirola, Pyomo – Optimization Modeling in Python, Second ed., vol. 67, Springer, 2017.

[3] W. E. Hart, J.-P. Watson and D. L. Woodruff, "Pyomo: modeling and solving mathematical programs in Python," *Mathematical Programming Computation ,* vol. 3, no. 3, pp. 219-260, 2011.

[4] D. J. Anderson, Kanban: successful evolutionary change for your technology business, Blue Hole Press, 2010.