Software Engineering Need Not Be Difficult

Jeffrey Carver, University of Alabama
George K. Thiruvathukal, Loyola University Chicago

September 8, 2013

Abstract

"Progress in scientific research is dependent on the quality and accessibility of software at all levels" (the overall premise of the workshop). We argue that true progress depends on embracing the best traditional–and emergent– practices in software engineering, especially agile practices that intersect with the tradition of software engineering. Software engineering as practiced today is more than the stereotypical monolithic lifecycle processes (e.g. waterfall, spiral, etc.) that historically have impeded progress for small/medium sized development efforts. In addition, the discipline and practice of software engineering includes software quality (with an established tradition of software *metrics*). Software processes can be pragmatic and use best features/practices of various models without impeding developer productivity. The embracement of these practices may also be important to prevent a brain drain of sorts, as students are increasingly eschewing traditional scientific/computation science research in favor of industry opportunities, where they can literally apply what they have learned in software development courses where pragmatic software engineering practices (e.g. test-driven design, RESTful architecture, etc.) are already prevalent.

Introduction

Because software engineering itself is a complex topic–and one that often evokes controversy–our goal in writing this paper is to combat a common, and often incorrect perception that Software Engineering has to, by definition, include only practices that are extensive, process-heavy and span the development lifecycle. Our thinking in this regard is influenced by efforts such as the Team Software Process and Personal Software Process methodoligies [1] and Agile methods [2]. Separately, we argue that if Software Engineering is viewed as a collection of practices that can be tailored and applied as appropriate rather than an all-or-nothing monolith, then it can make great contributions to the development of sustainable scientific software.

Based on our experiences, we realize that software engineering practices are only needed to the degree at which they are helpful to a particular project. That is, we are not advocating practices that should necessarily be followed by all teams. That said, we have also observed that many scientific/engineering projects discover that software engineering practices are necessary only after they have encountered a problem which cannot be easily solved. This situation often arises

[1] http://www.sei.cmu.edu/tsp/
[2] Agile Manifesto, http://agilemanifesto.org

when the source of the problem is the people rather than the technology. (A really good example is project management. It becomes vital when trying to operate at scale, by which we mean *human* scale.) In such cases, software engineering is an afterthought rather than a forethought. Adding software engineering practices late in a project tends to be more difficult and expensive than adding them early. Teams need to be willing to sacrifice some additional costs early to reap the benefits later.
In the remainder of this position paper, we discuss the software engineering practices that we have seen work in this environment. We also speak more generally about those practices, which we hope results in a checklist (one that can be updated, thanks to the guidelines of this workshop, which allow for versioning of this paper). We conclude with a summary of some recent efforts (mini case studies) by the authors on applying software engineering practices to real-world science/engineering projects at the national labs (Carver) and in bioinformatics (Thiruvathukal) with our respective collaborators.

How to Embrace SE via Lightweight and Agile Processes

In our experiences interacting with various scientific teams, we have observed a number of lightweight software engineering practices being employed. These practices all serve to make the software more sustainable either by operating directly on the code or by operating on the development process through the addition of structure. We provide that list here as a resource from which other teams may begin exploring.

- **Source code management** (a.k.a. version control) through Distributed Version Control Systems (Mercurial and Git) in the cloud allow teams to manage the evolution of the software and easily maintain multiple experimental and production versions as necessary.
- **Wikis** that provide lightweight documentation of the software (e.g. Google Sites, wiki provided by DVCS hosting solutions such as Bitbucket and GitHub)
- **Issue tracking** using cloud-based or open-source tools allow teams to track defects that must be fixed and features that must be added to the software.
- **Automatic build and release management** using *continuous integration* systems and/or build farms that simplify the process of building and releasing the software so those steps can be performed more frequently.
- **Project management**, i.e. lightweight task trackers like Trello (www.trello.com). We've talked about this trend in previous work [3].

We claim that most of these practices lead to sustainable outcomes for obvious reasons. Sustainable innovation is simply not possible without engendering a culture that is focused on preservation, which means not only the code but the rationale that went into creating it, the planning and execution thereof, and a culture of releasing "early and often" (a phrase used in the agile software development community). A key commonality among these practices is the ease with which they can be added into an existing development process as needed.
In addition to those practices, we have a second list of practices that we have seen much more infrequently (if at all) with our scientific collaborators. This next list serves more as a "wish

[3] David Dennis, Konstantin Läufer, and George K. Thiruvathukal, "Initial experience in moving key academic department functions to social networking sites", In Proc. 6th International Conference on Software and Data Technologies (ICSOFT) (July 2011)

list" of practices that we would like to see scientific teams begin to consider employing on their projects.

- **Unit testing**: Whether or not the development language has a unit testing framework, unit testing is the only way to build large-scale software today, and it is a widely taught and used practice throughout the industry, especially in web 2.0 and beyond. It is by far most effective when used in object-oriented languages (where frameworks such as JUnit [ˆjunit] were pioneered). Nevertheless, there are C projects that have shown how to do unit testing effectively n more *ad hoc* ways use it as well (e.g. the MPICH project) by running unit tests as processes whose results are checked in the shell, which has support for success/failure testing.

- **Test Driven Development**: A consequence of unit testing, TDD is where testing and development are concurrent activities. In practice, this does create a bit more work for the developer at first, but as code evolves, a good set of unit tests can evolve with it. In our own projects, it is incredible how many coding errors (when revising) can be caught by well- crafted unit tests.

- **High-level Requirements**: Nobody particularly likes writing documentation. Full requirements specifications are overkill, but using a wiki to document the key ideas and use-cases that went into creating software in the first place can help projects remain coherent. Piling on features that have nothing to do with the initial rationale for a project result in bloatware that will ultimately be displaced by simpler and more focused solutions.

- **Metrics**: a.k.a. *measurement* provides insight into the real problems faced by developers, should focus on those metrics that can have immediate impact on the development process. (e.g. defect density and issues reported with respect to time are almost trivial to incorporate in most projects, especially those hosted on services like Bitbucket and GitHub.

- **Documentation**: Can help save time during later rework, is facilitated by open-source tools like Sphinx [4] and Doxygen [5] In the case of Sphinx, it can be serve as a wiki and requirements capturing tool as well. This document itself is prepared using Markdown via the Pandoc [6] system.

- **Continuous Integration**: Can be combined with other practices like metrics and daily builds to ensure that bugs are caught quickly and tracked for later improvements. Examples are Jenkins [7] or Team City [8]. If a project does not build and pass unit tests, you'd like to know before your users tell you (this also has the effect of improving certain metrics related to defects).

- **Code review**: Many types of bugs can be more easily detected through peer code review than through extensive testing - may be some resistance because developers who consider themselves strong coders may not think review is necessary - but even the best coders make mistakes.

[4] http://sphinx.pocoo.org
[5] http://www.stack.nl/~dimitri/doxygen/
[6] http://johnmacfarlane.net/pandoc/
[7] http://jenkins-ci.org/
[8] http://www.jetbrains.com/teamcity/

- **Abstraction**: It's not just for computer scientists. It is a key component of modern software engineering. By separating data and methods that operate on it, software is more likely to be used and modified by others. There are some counterexamples in the real world but virtually all successful projects, even low-level ones such as Linux and MPI (Message Passing Interface), which were written in C, strongly embrace abstraction and therefore can be extended by others for purposes that were previously unimagined or envisioned.

Some Examples

To help illustrate the concept that software engineering does not have the be heavy-weight and difficult, this section provides examples of the use of lightweight software engineering practices in the development of CSE software for various platforms.

Peer Code Review

One of Carver's computer science PhD students focusing on software engineering spent two months in the summer of 2013 working with a computational science team at Oak Ridge National Laboratory. The goal of this interaction was to identify project needs that could be supported by lightweight software engineering practices. The hope was that this type of interaction could serve as an example of the successful use of appropriate, lightweight software engineering practices by a CSE team developing complex, parallel software.
The student had some initial meetings with the team to understand their most pressing problems and where he could best help them. After these meetings, the student determined that implementing peer-code review on the team would be the most beneficial direction to pursue. The development team was receptive to this suggestion and thought it would be helpful. The student then proceeded to train the team members on how to perform peer reviews. This interaction is still relatively early, so we do not yet have concrete results that can provide objective validation. But, the anecdotal data from the team members suggests the inclusion of peer code reviews has been positive. All team members have participated both as a reviewer and as a reviewee. Furthermore, the team leader has indicated that the use of peer code review resulted in the identification of a number of defects that would not have been revealed using their current testing methods. This result suggests that the use of peer code review has already had a beneficial effect on this project. Our data collection efforts over time will allow us to draw more objective and concrete conclusions about the benefits of peer reviews. One other positive result from the addition of peer code reviews is that it has motivated the team to implement a coding standard to make their code more readable. Interestingly, the decision to add a coding standard was made by the team members themselves rather than being suggested by the software engineering PhD student. Once the team made this decision, the software engineering PhD student helped them choose an appropriate standard.

Test-Driven Development

Another one of Carver's computer science PhD students is working on the application of Test Driven Development (TDD) to the development of CSE software. We have examples of two types of interactions, one that resulted from an extended visit, and is ongoing, and another that will be done remotely, and is just in the beginning phases.
In the first example, the PhD student spent a semester at the Combustion Research Facility of Sandia National Laboratories. During this time, he worked as part of the team developing the

Community Laser-Induced Incandescence Modeling Environment (CLiiME) [9] software. His goal in working with the team was to both serve as a developer of CLiiME as well as to study the use of TDD during its development. The team consisted of domain experts from Mechanical Engineering and Computational Chemistry. The student implemented a slightly modified version of TDD to support the development of this CSE software. As a results of the use of this approach, the software was successfully developed. We are in the process of conducting additional studies on this software to better quantify the benefits provided by using TDD to build the software. (Nanthaamornphong, et al.)

In the second example, we are beginning to work with a team developing an atomic microscopy code to be run in the Ohio Supercomputing Center. This team consists of two members who have a computer science background and two members who are domain experts. The interaction with this team will be different than in the previous example. Rather than participating on the team, we will serve as outside consultants/researchers. We will first provide the team with training on the use of TDD and be a resource for them as they use it. We plan to collect various types of quantitative and qualitative data to measure the impacts of using TDD and to tailor the process as needed. We anticipate that this project will result in an interesting case study that could serve to encourage other similar teams to experiment with TDD in their own environments.

Polyglot Thinking

Speaking further to TDD and pragmatic software engineering, Dr. Thiruvathukal and his team at Loyola University Chicago have been working on building a bioinformatics data warehouse to perform large-scale, multidimensional studies on the evolution of HIV and other viruses [10]. In the process of building this resource, we've been taking a polyglot approach to developing the scientific software itself. We built typesafe parsers for bioinformatics data sets in Scala to transform the data into one that more readily lends itself to analysis. We make extensive use of test-driven design with automated unit tests. Owing to the challenging nature of working with bioinformatics, we have recently incorporated a continuous integration server (Team City) to automatically build and run unit tests, so we know in real time whether anything with the software has failed. This is crucial because our warehouse keeps growing, and it is often the case that changes happen both to the parsing APIs.

Why Objects Matter

Architecturally speaking, we embrace the pragmatic use of object-oriented design. For example. Scala was extremely helpful for building a reliable parser–something that should be done with static type checking. When we built the web service, however, we found that we're largly working with flat tuple structures (of string data that has passed all type checks); Python proved a solid choice. Furthermore, the web services frameworks are a bit more stable in Python than Scala. Even so, both languages embrace test-driven design and unit testing, so we are able to stick to good software engineering without being pedantic about the language choices.

[9] Aziz Nanthaamornphong, Karla Morris, Damian W. I. Rouson, and Hope A. Michelsen "A Case Study: Agile Development in the Community Laser-Induced Incandescence Modeling Environment (CLiiME)" Proceedings of the 2013 International Workshop on Software Engineering for Computational Science & Engineering, May 18, 2013, San Francisco, CA.

[10] S. Reisman, C. Putonti, G. K. Thiruvathukal, and K. Läufer. "A Polyglot Approach to Bioinformatics Data Integration: Phylogenetic Analysis of HIV-1: Research Poster". 2nd Greater Chicago Area System Research Workshop (GCASR), May 3, 2013, Northwestern University, Evanston, IL, USA.

Conclusion

This paper has argued that there are a number of light-weight software engineering practices that scientific teams can use to make their software more sustainable without incurring a large amount of overhead. We provided two lists of practices: those that we have seen used effectively and those that we would like to see used more frequently.
We advocate that individual developers adopt light-weight practices that are beneficial to their own work (i.e. PSP, TDD, etc. . .). These are practices that could have significant benefit for software quality/productivity even if the entire team does not buy in. Working in this way will increase the sustainability of scientific software bottom-up with little institutional commitment required upfront to make progress.
As input to the workshop, we pose the following questions for audience discussion:

- What is the mix of projects people are doing? (ranging from small 1-2 person teams to larger teams) - this may affect which practices are useful or the degree to which the practice must be formalized.
- What software engineering practices are teams currently using?
- What software engineering practices have teams tried to use but not found much benefit in?