Barely Sufficient Software Engineering: 10 Practices to Improve Your CSE Software

Michael A. Heroux*
Sandia National Laboratories
maherou@sandia.gov

James M. Willenbring*
Sandia National Laboratories
jmwille@sandia.gov

Abstract

Computational Science and Engineering (CSE) software is typically developed using research funding where the primary focus is research and development of advanced algorithms and modeling capabilities. As a result, formal software engineering is seldom a primary goal. CSE software developers intend to write good software, but often lack the training, resources or time to adopt advanced formal methods and practices.

In this paper, we present a list of practices identified from the Trilinos project that we believe most CSE software teams can adopt and from which they can benefit.

1. Introduction

Computational science and engineering (CSE) applications can benefit from adoption of some commercial software engineering practices. However, in our experience, many CSE software developers have a skeptical view of formal software engineering practices, if they have any opinion at all. This impression comes from decades of working with CSE applications as part of introducing mathematical libraries into these codes and observing CSE software teams and their processes.

In this paper we discuss a small collection of practices that we believe can be most beneficial to CSE software projects. All of these practices come from our experience on the Trilinos project[1]. Some of them are very close to practices advocated by the Agile software development community [2], which shares some common needs with CSE. In fact, the term "barely sufficient" in our title is intended to reflect the Agile philosophy toward formality in general. By "barely sufficient" we mean that the practices we list here provide a respectable but minimal foundation for formal software engineering in support of CSE software projects. Certainly additional practices are valuable, but because of the nature of CSE software funding (which is provided to conduct science and engineering research and development, from which software is only one of the deliverables) a heavy emphasis on software engineering can be a distraction to the real project goals.

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

1.1. Research vs. commercial software

Commercial software, written for the purpose of generating revenue typically in domains where the underlying algorithms and methodologies are mature, has become increasingly sophisticated and complex, yet at the same time more easily developed and more reliable. Arguably this is primarily because software engineering is becoming a more mature field, with better-defined practices, more highly trained engineers and repeatable, predictable processes.

In contrast, research software, primarily in CSE disciplines, has as its main focus development of new algorithms and modeling capabilities. Software is developed as proof-of-concept and to generate first-of-a-kind results. Highly trained scientists, not professional software engineers, develop research

software. These scientists typically have little formal software engineering training but can often produce high quality software by using common sense principles and self-discipline. As a result, many CSE research software efforts generate high quality products, even though the software teams are largely unaware of standard industry concepts and practices. Even so, CSE software projects often apply basic software principles in an *ad hoc* manner that makes it difficult to leverage a product outside its narrowly intended scope.

In our experience, adoption of formal software practices by CSE projects is hindered by the fact that the software product is primarily a vehicle for producing science and engineering results, not an end goal itself and a general impression that too much formality can do more harm than good. As a result, practices must be introduced carefully so that generation of science and engineering results is not negatively impacted and formality is gradually increased.

As practices are introduced, the advantages of each practice should be stressed carefully to the development team. In our experience, it is important that team members believe that a new practice will be useful, otherwise there is a great risk that the practice will not be followed consistently.

The Trilinos project is large-scale software effort that has goals to develop state-of-the-art numerical libraries for CSE, while at the same time incorporate and adapt modern software engineering practices to improve our processes and products. In this paper, we reflect on our experiences with the intent to list and describe those practices that can have a broad impact on other CSE software projects.

The remainder of this paper discusses ten (actually eleven) practices that we consider extremely useful for any CSE software project. We recommend these practices, without reserve, to anyone who wants to have a better software product and spend less time creating it so they can spend more time doing science and engineering.

We complete this section by discussing "Practice 0" which we consider fundamentally important but so basic that it should not need to be recommended.

1.2. Practice 0: Manage source (the basics)

Since Trilinos software consists of libraries, its developers are often involved in the introduction of Trilinos capabilities into existing CSE applications. As a result, we often get exposure to the software practices of other teams. In our experience, the vast majority of CSE software projects use some form of source management. Specifically, source files for the project are kept on a common server where all team members can obtain a working copy and no one directly modifies the primary repository. However, we have seen numerous cases, especially with new projects, where developers do not use source management tools, and even a few mature projects where the source repository is hopelessly out of date with versions that developers are using on a daily basis.

Therefore, we mention that the single most important practice a software team can adopt is basic source management. By this we mean that source files are kept in a repository, developers regularly commit changes to the repository, and the repository contents are a vital resource to the project team.

2. The Ten Practices

2.1. Practice 1: Use issue-tracking software for requirements, features and bugs.

Issue-tracking software provides a logical collection point for information concerning bugs, features, and requirements. There are several strong reasons to use issue-tracking software, rather than simply keeping personal files, reminders or post-it notes. Specifically:

- Issues can be visible to the whole team.
- Issue-tracking software commonly provides the ability to prioritize issues.
- Establishing dependencies between issues provides the ability to break larger issues down into pieces, or see how different issues affect one another.
- The history of issues is kept in a searchable location for future reference.

The dependency-tracking feature that many issue-tracking systems support can be utilized in many ways. For example, a large deliverable might depend on a number of smaller feature enhancements. The

deliverable can be filed as one issue, and each of the smaller enhancements can be a separate issue upon which the large deliverable will depend, with different groups of people tracking the progress of each issue.

For several years, the Trilinos team has used its issue-tracking tool Bugzilla [9] to manage release efforts. A slightly simplified view of this process is that one bug (issue) is filed for the release of each package (Trilinos functionality is composed of tens of independent packages), and any specific issues blocking the release of the package block the package release bug. Then those package release bugs block a bug that is filed for the release of all of Trilinos. Once all of the bugs blocking the Trilinos release bug have been resolved, and the Trilinos level release process checklists have been completed, a release can be certified. Using Bugzilla to manage this process allows for a unified view of the outstanding issues blocking the release.

2.2. Practice 2: Manage source (beyond the basics)

Beyond basic source management, a repository can serve many useful purposes if carefully used. There are many source management tools available, including SVN[3], CVS[4], and git[5]. Common concepts in repository management include tagging and branching.

Before a release, it is useful to branch the repository. Branching creates an independent line of development, separate from the standard development, or head branch. Changes can then be made to stabilize the release branch while continuing new development on the head branch. Changes appropriate for multiple branches can be merged from one branch to another.

A tag is a snapshot of the current state of the repository. Tags are commonly used to create a bit-wise identifiable release, to mark a point of departure before a new development effort, or to create a snapshot after changes are merged from another branch so that the start of the next set of changes to be merged is easily retrievable. Bit-wise identifiable releases are very important because they eliminate ambiguity when dealing with software faults. If a user has a problem with your product, you know exactly what source code generated the problem and you can provide them with a new version that is also uniquely different.

The distinction between branches and tags is that a branch is a new line of development and can be modified. Tags are snapshots along a line of development, and are not modifiable. Some version control systems do not use tags, but rather use branches for tags and branches. It is then up to the development team to respect the concept of a tag and not modify that branch.

Some source management tools, including SVN and CVS, have associated source browsing and viewing tools. ViewVC[6] is a tool that can be used with SVN or CVS. Bonsai[7] is a tool that is compatible only with CVS. Bonsai can search the repository for all revisions based on user, branch, filename, date, and CVS module, as well as other criteria. It is also possible to browse through the project directory structure to find current and historical files and see the revision history for those files. Any two versions of files can be compared for line-by-line modifications.

2.3. Practice 3: Use mail lists to communicate

Mail lists allow for simple and effective communication. There are numerous advantages to using mail lists for a CSE software project. Rather than sending a message about a project to a personally selected group of recipients, mail lists allow interested recipients to self-identify. Using a centralized mail list tool instead of keeping personal lists of interested recipients prevents the lists from getting stale - new developers and users are not forgotten, and former developers and users do not continue getting irrelevant messages. Several different mail lists are appropriate for many projects including:

- Users – Communication amongst users and between users and developers. Commonly used as a trouble-shooting list.
- Developers – Communication amongst and important announcements for developers.
- Leaders – Communication amongst and important announcements for project leaders (including a subset of the developers and management or other key stakeholders).
- Regression – Automated messages containing test harness results.
- Check-in – Commit messages pertaining to code modifications. Messages to this list should be

automatically generated from commit logs of the source management tool.
- Announce – Announcements (often for releases or new features).

Mail lists are also useful for archival purposes and spam filtering. We have found Mailman[8] to be a useful mail list tool.

It is worth noting that wikis may be used in addition to mail lists. Wikis have the advantage of hypertext browsing, real-time editing and collaborative development of content. However, they are not a replacement for mail lists. Directed content delivery by email, and archiving of mail list messages are critical capabilities.

2.4. Practice 4: Use checklists for repeated processes

Checklists are valuable tools for making easily repeatable processes and for training purposes. The Trilinos project uses several different checklists, including a variety of release checklists, a new developer checklist, and a CVS commit checklist. Completing each item on the release checklists makes it much easier to remember an important, but easily omitted, step such as posting the documentation on the website for the latest release version, or updating the list of changes for the current minor release.

When training a new developer, a checklist can help to make sure that the developer is familiar with all of the tools and software used by the project, and that common team practices are shared with the new team member. Without proper training, it is easy for a new developer to omit an important test, for instance, and revise the code base without following the proper policy.

2.5. Practice 5: Create barely sufficient, source-centric documentation

As mentioned in the introduction of this paper, the term "barely sufficient" reflects a minimalist attitude to formal processes, adopting only those that have a large impact. In a similar way, documentation should be sufficient but minimal. In our experience, one of the biggest mistakes a CSE software project can make is to adopt large-scale formal document generation in a project that is just starting to focus on explicit software engineering practices. These documents require a large effort, much of which is often done to satisfy an external requirement, and does not benefit the project team. Furthermore, these documents become out-of-date quickly and therefore are irrelevant or even misleading.

Instead we have found that a combination of near-to-the-source and in-source documentation can be very effective. Specifically we find that the following approaches work well:

- User-callable functions and executables should be documented in the source files, using minimal markup such as that found in Doxygen[10]. Processing source files then generates documentation. This approach makes it much easier to keep documentation up-to-date.
- Higher-level conceptual documentation should be custom-developed, but still tightly coupled to examples that are part of the software repository. As much as possible, examples in the documentation should be extracted from actual working examples in the repository.
- Requirements, analysis and design documentation should be captured by appropriate tools such as Bugzilla (for requirements) and UML graphics tools (e.g., Microsoft Visio). Tools like Doxygen can also be used for design discussions since they produce UML diagrams directly from source code. Documentation efforts should not result in long hand-written, text documents until a project reaches a level of maturity where there is little change in software design and implementation.

Unfortunately, for many software teams, their first experience with formal software engineering is an imposed requirement to produce formal, detailed requirements, analysis and design documentation that adds little value to a CSE software project and distracts developers from important science and engineering work.

Formal documents certainly play a role in a project, but should be developed after the product architecture is stable. Formal documents are essential when a product is ready for hand-off to a maintenance team that is not the original scientific development team.

2.6. Practice 6: Use configuration management tools

The use of configuration management tools can make software accessible to a much broader audience and make software support much less expensive. Building software using hand-written makefiles, which is very common for CSE software, is challenging for a large percentage of users. Providing a simpler method of installation, such as a CMake-based[11] build system, or better yet, a Linux RPM or Windows installer will not only cut support costs for existing users, but will also make the software available to a group of users who previously chose not to take the time to complete a complicated installation process.

CMake in particular is very portable and supplies a rich set of build targets. The benefit of a CMake build system will far outweigh the cost. Tools like Cmake are trivial to use for simple projects and lead to minimal overhead. For any code that requires more than a simple, portable set of commands for installation, configuration management tools are challenging to adopt, but provide tremendous value in the long run.

2.7. Practice 7: Write tests first, run them often

Testing is essential for any high-quality software product, but many CSE developers view tests as something that should be developed late in the software development process since that is when a product is available for testing. In our experience, we find that the philosophy of test-driven development [12] (TDD) is very valuable. TDD means that developers write tests first, before the software product is written, and provide a full coverage of the functionality that the product is expected to deliver.

Writing a collection of tests first has a number of benefits:

- Software test programs debug your design because they mimic how the user will interface with your product. In this way your design is validated to some extent before implementing the product.
- Although initially all your tests will fail, as your software product is developed, an increasing number of tests will pass, giving you a measure of how close you are to completing your implementation.
- A full suite of tests provides you with confidence to revise your software after the initial implementation and improves the long-term quality of your product as it matures.

Adopting TDD as a habit can be a cultural challenge, since writing the tests delays the initial development of source code. But in our experience it provides tremendous value by greatly reducing development costs and improving long-term software quality.

2.8. Practice 8: Program tough stuff together

Pair programming is a concept formalized by Extreme Programming [13]. This approach to software development means that two people sit together and develop software. In our experience, this practice is not natural for CSE developers, who are more used to sitting by themselves to carefully write source code. Therefore, we do not advocate pair programming for all development. However, we have found that for development of complex software functions, working with a partner side-by-side is very valuable. This is especially true for situations where one developer is incorporating the use of another developer's software. In this situation, having the second developer act as a "navigator" for the first developer provides value to both developers. The activity produces superior software and provides important feedback to the second developer.

2.9. Practice 9: Use a formal release process

When combined with continual process improvement (Practice 10), following a formal release process is an invaluable practice for a software team. When a software project is just getting started, an appropriate release process may simply be to run some reasonable set of tests on a defined set of platforms, and tag the new version when all of those tests pass. Even in a simple case, verifying that the test suite runs on supported platforms and making sure that a released version of the code is bit-wise identifiable makes user support much more manageable and efficient.

For larger software projects, a formal release process is essential, not only for reaching a stable point at which a release can occur, but also for managing the process in a controlled way so that when all necessary processes have been completed, a release can be completed with greater confidence.

As Trilinos and its user base have grown, the release process for a major release has gone from an

informal series of tests on a release branch to a much larger, coordinated effort. In addition to Trilinos level testing, we work with multiple key users to certify their test suite against the release candidate. After each release, the processes are reviewed for ways to improve the next release.

Completing the entire major release process for each minor release (typically providing bug fixes or very small enhancements) does not provide enough benefit to justify the cost, so a subset of the major release process is used. This carefully chosen subset is periodically evaluated for effectiveness, and to consider significant changes, such as the availability of additional automated testing results from key user applications.

2.10. Practice 10: Perform continual process improvement

Improving software processes is an on-going effort. Any software process, no matter how poorly defined, can be written down and improved upon, and any process, no matter how mature, can be made better.

Consider the process of training a new developer. Depending on which team member is conducting the training, what training takes place can vary greatly. Until a draft process is recorded, the training used will be haphazard, based on what the trainer happens to remember. By standardizing the training with a checklist, consistent training that touches on the most important aspects of the job can be given to each new team member. Even if the initial checklist is lacking some items, at least some items will be covered, and missing items or new items will be added over time through conscious process improvement.

Every time a checklist is used, the user should consider whether or not modifications are necessary. Having several people use the same checklist allows the opportunity to combine all of the best ideas into one standard list.

Another important aspect of process improvement is to include items on process checklists that reflect future goals, rather than current requirements. For example, one optional item on a package release checklist could be measuring the code coverage provided by the test suite. In the future, a project could compute code coverage for all releases, but by including the item on the current checklist, we ease the transition if measuring code coverage becomes a requirement, and provide concrete evidence of process improvement.

3. Conclusions

CSE software can benefit from modern software engineering practices and processes. At the same time, because the goal of CSE software is often research and development such that the software product is just one output, too much emphasis on software processes can put a project at risk. The 10 practices we present in this paper should not require a large effort for most CSE software teams and, once adopted, should provide a qualitative improvement in the overall software development process, producing better quality software with less effort and giving CSE project teams more time for science and engineering research and development.

4. References

[1] M. A. Heroux, "Trilinos Home Page", http://trilinos.sandia.gov, 2009.

[2] "Agile Software Development Home Page", http://www.agile-software-development.com, 2009.

[3] Tigris.org, "Subversion Home Page", http://subversion.tigris.org, 2009.

[4] "Concurrent Versions System Home Page", http://www.nongnu.org/cvs, 2009.

[5] Scott Chacon, "Git – Fast Version Control System Home Page" http://git-scm.com, 2009.

[6] Tigris.org, "ViewVC Home Page", http://www.viewvc.org, 2009.

[7] Mozilla, "Bonsai Project Home Page", http://www.mozilla.org/projects/bonsai, 2009.

[8] GNU, "Mailman, the GNU Mailing List Manager Home Page", http://www.gnu.org/software/mailman, 2009.

[9] Mozilla, "Home::Bugzilla::bugzilla.org Home Page", http://www.bugzilla.org, 2009.

[10] Dimitri van Heesch, "Doxygen Home Page" , http://www.stack.nl/~dimitri/doxygen, 2009.

[11] Kitware, "Cmake - Cross Platform Make Home Page", http://www.cmake.org, 2009.

[12] K. Beck, *Test Driven Development: By Example*, Addison-Wesley, Boston, 2003.

[13] K. Beck, *Extreme Programming Explained*, Addison-Wesley, Boston, 2005.