International Journal of Current Science Research and Review
ISSN: 2581-8341
Volume 08 Issue 01 Jan 2020
www.ijcsrr.org

Simply Sufficient CSE Software Engineering: 10 Best Practices

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

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

Abstract

In most cases, the creation of sophisticated methods and modeling skills is the main emphasis of study financing for Computational Science and Engineering (CSE) software. That's why most projects don't make traditional software engineering a top priority. Good software is written by CSE programmers all the time; they just don't always have the time, money, or formal education to implement it. This document presents a set of best practices for CSE software development teams that we've gleaned from the Trilinos project.

1. Introduction

Adopting some business software engineering techniques can improve computational science and engineering (CSE) apps. Our observations suggest, however, that many CSE programmers regard traditional software engineering techniques with skepticism, if they have an opinion about them at all. Decades of integrating mathematical tools into CSE apps and watching the workflows of CSE software teams have left me with this perception.

This article focuses on a handful of methods that we think can significantly improve CSE.

PREPRESS PROOF FILE

programming initiatives. All of these methods were honed during our work on the Trilinos project[1]. The Agile software development group [2], which has some requirements in common with CSE, advocates for some of these same approaches. Indeed, we chose the word "barely sufficient" for our title so that it would convey the Agile perspective on decorum. When we say that the procedures we've outlined here are "barely sufficient," we mean that they form an acceptable but minimum basis for formal software engineering in aid of CSE software initiatives. The nature of CSE software funding (which is provided to conduct science and engineering R&D, of which software is only one of the deliverables) means that a heavy emphasis on software engineering can be a distraction from the real project goals, despite the value of additional practices.

1.1. Research vs. commercial software

This trend can be seen in the growing sophistication and complexity of commercial software, which is usually created for profit in areas where the fundamental methods and techniques are well-established. This is largely attributable to the fact that software engineering is maturing into a discipline with established norms, expert practitioners, and reliable procedures.

In comparison, the primary emphasis of study software, especially in CSE fields, is the creation of novel algorithmic and modeling capabilities. Creating original software is done so as a demonstration of idea and to produce novel outcomes. Research is created by highly educated experts, not by experienced software developers.

CAUSAL PRODUCTIONS

software. Despite their lack of software engineering instruction, these experts are often capable of creating high-quality software through the application of common sense principles and self-discipline. So, even though many software development teams working in the field of computer science and engineering (CSE) are unfamiliar with common industry ideas and practices, they often produce high-quality products. However, fundamental software principles are often applied haphazardly in CSE software initiatives, making it challenging to utilize a product beyond its tightly intended purview.

As far as we can tell, the main obstacles to the widespread use of formal software practices in CSE projects are the widespread belief that too much formality can do more damage than good and the reality that the software product is mainly a tool for generating scientific and engineering results rather than an end objective in itself. Therefore, it is important to progressively increase discipline while ensuring that the production of scientific and technical outcomes is not hindered by the change.

The benefits of each new practice should be meticulously emphasized to the development team as they are presented. Our research shows that if team members do not think a new practice will be beneficial, it is highly unlikely that it will be regularly implemented.

Large-scale software endeavor Trilinos aims to better our processes and products by developing cutting-edge math tools for CSE and adopting and adapting contemporary software engineering practices. In this article, we look back on our work and try to identify the techniques that could have a significant effect on other CSE software initiatives and explain them. Ten (technically eleven) best practices for developing CSE software are discussed in the following sections. Without hesitation, we endorse these methods to anyone seeking to improve the quality of their software products while decreasing development time in order to devote more resources to actual science and engineering.

Finally, we cover "Practice 0," which is so essential that it shouldn't even be suggested but which we think is crucial nonetheless.

1.2. Practice 0: Manage source (the basics)

Since Trilinos is a library, its developers are often requested to add the framework's capabilities to other CSE software. This means that we are constantly learning about the processes used by other teams to create software. We have observed that the vast majority of CSE software projects use some form of version control. Specifically, the team members can view a working copy of the project's code files on a central machine, but they are not permitted to make any modifications to the master location. However, there are many cases where developers do not use source control tools, especially in newer projects, and there are also some earlier projects where the source file is totally out of date with versions that developers actively use on a regular basis.

We stress the importance of basic source control because it is the cornerstone of any successful software development team. This indicates that the project uses a repository to house its source code, that developers frequently submit changes to the repository, and that the contents of the repository are heavily utilized by the project team.

2. The Ten Practices

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

Using issue-tracking software, you can keep track of all your bugs, features, and requirements in one spot. There are many good reasons to use problem-tracking tools rather than depending on your own files, reminders, or post-it notes. As a whole, we can see the issues that need to be addressed. The ability to prioritize issues is a common function of problem tracking programs. Complex problems can be broken down into their constituent parts and their interdependencies examined through the establishment of connections between issues.

All issues are documented in one easily accessible location.

Many booking systems include a dependency-tracking feature that can be used in a variety of contexts. One significant release may rely on the successful conclusion of several smaller feature improvements. While the product itself is submitted as a single issue, the various enhancements that go into making that release possible can be submitted as separate issues and monitored by separate groups.

Bugzilla [9] has been used by the Trilinos team for several years to track problems and organize delivery efforts. The distribution of each individual package (Trilinos capacity is made up of scores of individual packages) is tracked by a single defect (issue), and any problems with releasing a package are flagged as such. Then, the flaws in the product versions stop us from filing a bug complaint for Trilinos as a whole. Once all problems blocking the Trilinos release have been resolved and the process for releasing to the Trilinos level is complete, the release can be confirmed. Using Bugzilla for collaboration paints a clearer image of what issues are still holding up the release.

2.2. Practice 2: Manage source (beyond thebasics)

A repository's utility extends far beyond that of simple source control. In addition to CVS[4] and git[5], SVN[3] is also an option for code control. Tagging and splitting are commonplace in file administration. It's helpful to split the file before releasing. When a line of growth splits off from the main line, or head branch, it is said to have "branched." Once the release branch has been stabilized, fresh work can continue on the "head" line. It is possible to combine changes that are applicable to numerous lines into a single branch.

The repository's present state is frozen in time as a tag. To quickly find the beginning of the next set of changes to be combined from another branch, developers often use tags to create a copy after changes are merged from another branch. Tags can also be used to create a bit-wise recognizable version. Having versions that can be identified bit by bit removes any doubt when troubleshooting software issues. If a customer experiences an issue with your product, you can pinpoint the offending line of code and issue a completely new build.

A branch is a separate growth path that can be altered, in contrast to a tag which cannot. Tags are immutable, static points in time along a timeline of growth. Tags are not required in some VCS implementations; instead, branches are used in place of tags. The onus is then on the development team to adhere to the tagging idea and refrain from making any changes to the marked branch.
SVN and CVS are two examples of source control programs that come with complementary source reading and watching applications. ViewVC[6] works with both Subversion and CVS. The Bonsai[7] application is CVS-exclusive. Bonsai allows you to look for specific changes in the source by using filters such as user, branch, files, date, and CVS module. Files, both new and old, and their modification histories can be viewed by navigating the project's root hierarchy. Changes between two files of any edition can be examined line by line.

2.3. Practice 3: Use mail lists to communicate

Mailing lists facilitate efficient and straightforward correspondence. Using mailing groups for a computer science and engineering software endeavor has many benefits. Instead of delivering a communication about a project to a small, hand-picked group of people, interested parties can identify themselves via mail groups. A unified mailing list utility eliminates the need for individuals to maintain their own mailing lists of interested parties, ensuring that neither new developers nor users are neglected, nor do inactive developers or users continue to receive messages that are no longer pertinent to them. There are a variety of mailing groups that can be useful for a wide range of tasks.
Users - User-to-User and User-to-Developer communication. Used frequently as a checklist for fixing problems.
Developers can talk to one another and learn crucial updates in this section.

Leaders - Interaction with other leaders and critical updates for project managers (including a subset of the developers and management or other key stakeholders).
• Regression - Messages sent automatically with the findings of the test framework.
• Check-in - Messages indicating changes to code have been committed. Emails sent to this group should

produced mechanically from the source control software's change records.
Announcements (typically for product launches) fall under the Announce category.
Archiving emails and blocking unwanted messages are two additional uses for mailing lists. Mailman[8] is an effective mail list utility that we have used.

Wikis can be used as an alternative to email groups, which is something to keep in mind. Wikis' strengths are in the areas of shared content creation, real-time updating, and hyperlink navigation. But they can't substitute mailing groups. Important features include mail list message storage and directed content distribution via email.

2.4. Practice 4: Use checklists for repeatedprocesses

When it comes to teaching new employees or improving existing procedures, checklists are invaluable. Various version checklists, a new developer checklist, and a CVS submit checklist are among the many used by the Trilinos initiative. By checking off each item as it is completed, release schedules make it much less likely that a critical but easily forgotten action, like revising the inventory of changes for the current minor release, will be forgotten.

To ensure that a newly hired worker is properly trained and is up to speed on the project's tools and software, a schedule can be used during onboarding. Without the right guidance, a new programmer may make critical mistakes, such as skipping an essential test or making changes to the code that violate company policy.

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

The word "barely sufficient," as stated in the opening to this paper, represents a limited approach to formal processes, in which only those with a large effect are adopted. The same holds true for the paperwork, which should not be excessive. Adopting large-scale formal document production at the outset of a project that is just beginning to concentrate on explicit software engineering practices is, in our opinion, one of the greatest errors a CSE software project can make. These papers call for a lot of work, most of which is done to please an outsider but does not help the project team. In addition, the passage of time renders these records obsolete and potentially deceptive.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:
• Minimal notation, like that found in Doxygen[10], should be used to describe user-callable methods and executables in the source files. When raw files are processed, paperwork is produced. Using this method, updating instructions is a breeze.
• Conceptual guidance at a higher level should be created from scratch, but it should still be closely linked to samples found in the software's central source. It is recommended that samples used in the instructions come directly from the source whenever feasible.

analysis, and design data. (e.g., Microsoft Visio). UML models can be generated automatically from source code using tools like Doxygen, making them useful for design talks. Until a project achieves a mature state where there is little change in software design and execution, it is not appropriate for documentation efforts to end in lengthy hand-written, text papers.
Many software teams' first exposure to formal software engineering is the imposition of overly detailed requirements, analysis, and design documentation that serves little purpose in a CSE software project and diverts developers' attention from the crucial science and engineering work they should be doing.
The development of formal papers is important, but it should wait until the product design is solidified. When it's time to pass off a product to a support team that wasn't involved in its initial scientific creation, formal papers are crucial.

2.6. Practice 6: Use configuration management tools

Using configuration management tools can make software more widely available and reduce the cost of software maintenance. It can be difficult for many people to build software from scratch using makefiles, which is standard practice for computer science and engineering software. Providing an easier installation method, like a CMake-based[11] build system or, even better, a Linux RPM or Windows installer, will not only reduce support costs for current users, but will also make the software accessible to users who previously avoided it due to its perceived complexity.

Particularly useful is CMake's portability and the abundance of build options it provides. A CMake build system's advantages will significantly exceed its disadvantages. For straightforward tasks, using a tool like Cmake is a breeze that results in negligible extra work. Configuration management tools are difficult to implement, but they are extremely useful for any code that needs more than a small, movable collection of instructions to deploy.

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

Many computer science and engineering (CSE) programmers believe that creating tests is a task best left until the end of the software development cycle, when a finished product can be tested. Test-driven development [12] (TDD) is a development methodology that has proven to be very useful in our experience. Test-driven development (TDD) is an approach to software development in which tests are created in advance of the actual software code to ensure that all anticipated features are covered. There are many advantages to creating a set of tests first:
• Software testing tools help you find and fix bugs in your code by simulating real-world user actions. By doing so, you can ensure that your plan will work before actually making the product.
Despite the fact that at first all of your tests will fail, the percentage of successful tests will increase as your software product is created.
• A comprehensive set of tests gives you the assurance to make changes to your software after its original execution, which in turn boosts the product's quality over time.

Writing the tests can be a societal barrier to TDD adoption because it slows down original source code development. However, it has proven to be extremely useful for our company, cutting down on expenses while simultaneously enhancing software quality for the long haul.

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 toboth 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 processimprovement (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.

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 processimprovement

Constant work goes into bettering software development procedures. No matter how rough your software process is in its current form, you can document it and make it better.

Think about how a novice programmer is brought up to speed. Training content can differ widely depending on which team member is providing instruction. Until a writing process is documented, any instruction provided will be random at best. Each new member of the team can receive uniform instruction that covers the essentials of their role by using a standardized schedule. Some things will be addressed even if they aren't on the original inventory, and the rest will be added gradually through deliberate process development.

Each time a schedule is used, the person utilizing it should evaluate whether or not any changes need to be made. By having multiple individuals use the same criteria, we can pool our finest ideas into a single document.

Including future objectives and standards on process plans is another crucial part of process development. Measuring the test suite's code coverage is an example of a non-obligatory item that could be included in a package delivery schedule. The potential for an undertaking to code coverage for each release, but by having it as an option on the existing criteria, we can more easily make the switch if that becomes a necessity, and we can show that the process has improved in the process of doing so.

3. Conclusions

The application of contemporary software engineering methods and procedures to CSE software would be beneficial. However, putting too much stress on software processes can be risky for a project because the aim of CSE software is often research and development and the software result is just one outcome. The ten practices we outline in this paper shouldn't require a lot of work from most CSE software teams, and once implemented, they should lead to a significant improvement in the quality and efficiency of the software development process as a whole, allowing CSE project teams to devote more time to scientific and technological innovation.

4. References

According to [1] M. A. Heroux's "Trilinos Home Page" (http://trilinos.sandia.gov), published in 2009.
http://www.agile-software-development.com/home.html (2009, Feb. 2) [2] "Agile Software Development."
The "Subversion Home Page" can be found at http://subversion.tigris.org/ (as of 2009) [3].

In 2009, http://www.nongnu.org/cvs was the home page for the Concurrent Versions System.
In 2009, Scott Chacon published "Git - Fast Version Control System Home Page" on the website http://git-scm.com.
ViewVC Home Page, Tigris.org, http://www.viewvc.org/, 2009.
In 2009, Mozilla published the "Bonsai Project Home Page" at http://www.mozilla.org/projects/bonsai.
To learn more about GNU Mailman, visit http://www.gnu.org/software/mailman. [8] GNU, "Mailman, the GNU Mailing List Manager Home Page", 2009.
For more information on Mozilla's issue tracking system, visit http://www.bugzilla.org/ (2009, titled "Home::Bugzilla").
[10] Home Page for Doxygen, Dimitri van Heesch, 2009, http://www.stack.nl/dimitri/doxygen.

According to Kitware's "Cmake - Cross Platform Make Home Page," available at http://www.cmake.org as of 2009 [11].
In 2003, Addison-Wesley published Test Driven Development: By Example by Kevin Beck.
Extreme Programming: An Explanation by Kent Beck, Addison-Wesley Publishing Company, Boston, 2005[13].