

Improving the Development Process for CSE Software

Authors:

Michael A. Heroux
James M. Willenbring
Michael N. Phenow

Computational Math and Algorithms Department,
Sandia National Laboratories, Albuquerque, NM, USA
E-mail: {maherou, jmwille, mnpheno}@sandia.gov

Abstract

Scientific and engineering programming has been around since the beginning of computing, often being the driving force for new system development and innovation. At the same time a continual focus on new modeling capabilities, and some apparent cultural issues, find software processes for many computational science and engineering (CSE) software projects lacking. Certainly there are notable exceptions, but our experience has been that CSE software projects, although committed to writing high-quality software, have few if any formal software processes and tools in place, and are often unaware of formal software quality assurance (SQA) concepts.

Presently, increasing complexity of applications and a broad push to certify computations are dictating a higher standard for CSE software quality; it is no longer sufficient to claim to write high quality software. However, traditional software development models can be impractical for CSE projects to implement. Despite this, CSE software teams can benefit by implementing valuable SQA processes and tools. In this paper we outline some the processes and tools that are successfully used by the Trilinos Project. These tools and processes have been useful not only in increasing verifiable software quality, but also have improved overall software quality, and the development experience in general.

1 Introduction

The Trilinos Project, located primarily at Sandia National Laboratories, is an effort to develop parallel solver algorithms and libraries within an object-oriented software framework for the solution of large-scale, complex, multi-physics engineering and scientific applications. Trilinos

consists of about thirty packages. Each package is focused on important, state-of-the-art algorithms in a particular domain and is developed by a small team of experts.

Four years ago the Trilinos project was charged with the task of improving its existing software quality practices. Since many computational science and engineering (CSE) software projects are not dedicated to formal Software Quality Assurance (SQA) practices, there is not a large body of literature that we can directly leverage and a lot of work is required to define practices that are well-suited to the project.

A few characteristics of the project make defining formal practices and processes especially challenging. Specifically, the requirements of the project are multi-faceted, both local (often defined at the package level) and global, and evolving, which makes it exceedingly difficult to maintain a universal formal requirements document. Also, the Trilinos team is geographically distributed, so we have a special emphasis on enabling effective methods of distributed communication. Finally, as with many CSE software projects, the budgetary focus is on algorithms development, leaving little money to put directly towards software quality efforts.

Now, four years into the process of improving software quality practices, we have found there are certain low-cost, high-yield processes and tools that tend to work well to enable the development of high quality CSE software. To illustrate this, we present some high-level goals that apply to most CSE software projects, and Trilinos in particular. We then present some principles that keep the Trilinos project on the right development path. Finally, we discuss some general classes (and specific instances) of tools that, guided by our driving principles, help us achieve project goals.

To learn more about the Trilinos project, visit the Trilinos web site [5].

2 Goals

Given the above project characteristics, we outline the high-level project goals. These goals can be applied to most software projects, but are described here in the specific context of the Trilinos Project.

2.1 Quality

Trilinos, like all software projects, seeks quality as a primary goal—quality both in the colloquial meaning of the word and the particular meaning it carries in the software engineering world, specifically: *a measure of the degree to which software meets its stated and implied requirements*. Additionally, for an increasing number of CSE software projects, claiming to do a good job or anecdotal user opinions of high quality is no longer sufficient. Customers increasingly require documented software processes.

2.2 Modularity

New solver and support capabilities in Trilinos are introduced as individual, autonomous modules called *packages*, developed by individuals or small teams. Keeping logically distinct pieces of functionality separate is critical to the long-term growth and health of the project.

2.3 Interoperability

Because of this modular architecture, it is of utmost importance that the various packages interact well together. For Trilinos to realize its full potential, all of the components need to work together in concert. This is an important issue for CSE software in general. A lot of excellent existing software is under-utilized because it cannot readily be integrated with other existing software and brought to bear on a single problem.

2.4 Scalability

Trilinos started as a collection of three packages. In a few short years, it has grown organically to include roughly 30 packages, and continues to grow. To maximize the benefits reaped from economies of scale and to leverage the power of other codes, scalability (in this context, the ability to continue to add more packages) is a primary concern for Trilinos. The degree to which the Trilinos architecture scales is directly dependent on the level of modularity and interoperability achieved. Another key scalability issue for Trilinos is that as packages are added, users should be shielded from the additional complexity; using Trilinos should not become significantly more complicated as Trilinos grows, outside of the complexity that a growing array of functionality inherently contains.

2.5 Efficient Use of Expert Time

Trilinos packages are developed by experts in the particular domain of a package. One critical goal of the Trilinos project is to make efficient use of these experts' time. These developers ought to spend as much time as possible working within their domain of expertise, leaving the vast majority of the software project management tasks to those who are specialists in that domain. The package domain experts should, however, provide input when selecting SQA processes, because adopting processes that are not well-suited to a particular project can decrease, rather than increase, efficiency.

2.6 Accessibility and Support

Finally, the ultimate goal for any piece of software is to actually have it be used. In order for people to use it, the software has to be reasonably accessible to users. It is also important to provide support so that all of the energy spent developing the software is put to good use, but, here again, it is important that the experts do not have to spend all of their time helping users install and use the software.

3 Driving Principles

As described in Section 1, Trilinos has a unique set of characteristics that make it differ from more common business software projects. However, by acknowledging these differences, goals can be formulated for the project. In our ongoing attempts to achieve our goals, Trilinos has been guided by a small set of principles that help the project stay on track when faced with critical decisions.

3.1 Package Orthogonality

Trilinos was originally created as a way to bring together parallel solvers to enable effective reuse and interoperability, and minimize duplication of effort by solver developers. The mechanism chosen for containing a solver was a “package.” A Trilinos package is simply a self-contained piece of software that is developed in the Trilinos source repository, can build within the Trilinos build system, and can interact with other Trilinos packages. The Greek word “trilinos” loosely translated means “string of pearls.” The name is meant to convey the idea that each package is independently valuable, and even more so when combined with other packages. This image also contains the notion of a common thread holding all the packages together.

Having both a collection of packages and a central entity gives Trilinos a two-tiered architecture. What we think of as the lower level is simply the packages. Above that, we have what we call the Trilinos framework. The framework

is where we seek to capitalize on economies of scale by providing global services to packages so duplication of effort is minimized wherever possible. The two driving principles that keep this two-tiered architecture in its delicate balance are “global services,” which we will discuss in the next section, and “package orthogonality.”

One critically important driving principle of the project since its inception has been preservation of package autonomy. While most packages do not stray far from the pack in terms of tools and processes, and may never do so, guarding autonomy has served us well. This principle of package autonomy has evolved into the more encompassing principle of package *orthogonality*. In this context, achieving orthogonality means that the relationships between different packages, as well as the relationships between packages and the framework, are such that a change has a minimal effect on other components.

Achieving a high degree of orthogonality is advantageous for many reasons. For instance, it allows packages to be effectively developed by small groups of domain experts without unnecessary hassles external to the algorithmic problem at hand. As mentioned above, Trilinos began as three packages and has since grown to include roughly 30. Each package has a unique history. Some were started from scratch within Trilinos. Others were existing projects imported into Trilinos. Of these, we find the whole range, from those just off the ground, to mature codes that have been in use for years. In all cases, the development of the code is done by experts in the given domain. These groups generally consist of one to five developers. This small size keeps the groups focused, agile, and accountable.

Packages that join Trilinos after they are relatively mature often would not do so if they felt they would be forever dependent on Trilinos. The design of the Trilinos architecture very deliberately seeks to prevent a central entity upon which all packages must be dependent. Many packages came to Trilinos already having an established user base and it is very important for some packages to be able to exist either within the Trilinos framework or completely apart from it.

Similarly, many packages would not be inclined to become a part of Trilinos if they had to surrender the control of their package. Being a part of Trilinos brings with it very few requirements. Instead, there are many guidelines and services that, in practice, are eventually adopted by all packages, *but at a pace determined by the package developers*. Local decisions about the direction or design of a package are left in the hands of the package developers.

Finally, maintaining a high level of package orthogonality and autonomy keeps us honest. Since packages teams are free to disassociate from Trilinos at any time, we know that to retain them (and thus to retain the benefits of the functionality they provide and the expertise of their devel-

opers) we must continue to provide value to the package developers.

3.2 Global Services

The Trilinos framework exists for the benefit of member packages, providing numerous services and suggested practices. The vast majority of the costs associated with implementing or adopting a given tool or service are constant and up-front. The cost of adding an additional package to Trilinos is usually negligible. This means that, on their own, the packages could not afford the time, energy, or expertise required to support such an array of services. By having the Trilinos framework provide these services, each package gains access to the whole suite of powerful services and tools—a high level of value at a cost that is effectively amortized across all packages.

Some of the standard services include source control, an issue reporting and tracking tool, and mail lists. More advanced services include a package webpage template, assistance in creating and maintaining package websites, and a build system that allows all packages to be built as a part of a single process and helps to ease porting issues. A functional example package called New Package can be used by developers to quickly adapt an existing piece of software to the suggested Trilinos build system or to hasten the process of developing portable software from scratch. The Trilinos test harness provides a framework for automated testing on a range of platforms, the ability to set up customized test runs, and the ability to view all test results online.

As mentioned above, Trilinos does not impose a large number of requirements on member packages. Rather, the Trilinos framework provides suggested practices that, with very few exceptions, are adopted by all packages. For example, packages are required to complete some sort of organized process prior to an external release (having a documented release process is a requirement that is imposed by powers above the Trilinos framework). The Trilinos framework team has developed a checklist that packages can complete to satisfy this requirement; however, package teams are free to develop an alternative process. At this time, every package uses the default release checklist, which saves developers the hassle of developing an individualized process and gives them a release process that has been hardened through process improvement based on feedback from member package teams.

3.3 Tight Collaboration

CSE software, like most software, has grown in complexity in recent years. The most interesting and challenging problems are generally not solved by an individual or project team working in isolation. Solid relationships with

external and internal collaborators are essential. It may even be the case that an outside collaborator is a significant stakeholder in the project and whose requirements are of utmost importance.

But how do you gather the requirements of your stakeholders? What happens when they change? Classical development models would prescribe a formal requirements-gathering process to set the direction of the project from the outset. From then on, all development has to be traceable back to the requirements and any deviation from the requirements warrants a formal revision of them.

For many CSE projects, this is not a reasonable approach. When your work, or the work of your stakeholders, is research-intensive or exploratory in nature, the problem may not be understood well enough at the outset of the project to make it worthwhile to define traditional formal requirements. Requirements will likely change and evolve very quickly. In such cases, attempting to adhere to a classical development model becomes unnecessary and time-consuming overhead instead of necessary bookkeeping.

How then to communicate effectively with your stakeholders? Establish a collaborative relationship with them. Bring them into the workings of your project. This does not mean that they have to be concerned with the day to day activities, but rather, use close collaboration to gather, implement, integrate, and iterate on their requirements. Proactively seek additional input from stakeholders on a regular basis and keep them well informed.

Trilinos encourages close collaboration amongst packages by establishing well-defined channels of communication. Issue-tracking software, mail lists, and regular meetings all give developers of one package the means to communicate with the developers of other packages to coordinate and create records of important design decisions.

Outside of the project, Trilinos maintains close relationships with its primary stakeholders, some of whom have a developer working on both Trilinos and the project in question. This helps ensure the successful integration of Trilinos into their codes. It also provides an effective means of staying abreast of the changing requirements of these external codes. Close collaborations of this nature help Trilinos prevent possible problems before they arise and also help to steer the project in the right direction.

3.4 Iterative Development

Close collaborations facilitate the communication of design decisions, requirements, and countless other important bits of information, but the ultimate goal of all this communication is to produce working code, and the longer development continues without being integrated and tested, the more time will have to be sunk into the integration and debugging processes. This has led the Trilinos Project to

strive for shorter iterations where possible. While this does not mean that we release as frequently as an aggressive Extreme Programming (XP) [11] project would, we are always looking for ways to shorten iterations in all areas of development.

More important than the time between iterations is the complexity between iterations. A complex iteration costs time, energy, and resources. When we minimize the cost of each iteration, we enable more iterations, and development can occur step by step, instead of in huge leaps and bounds. This makes the development more closely resemble extended rapid prototyping. Ideas are worked out in code, which then grows and matures organically into stable, robust software.

This principle of iterative, incremental development is valuable in a number of areas, from design, implementation, and debugging to building, testing, and integrating. In section 4 we will discuss a number of tools that the Trilinos Project relies on to enable short, simple, inexpensive iterations.

3.5 Process Improvement

Software processes are always a work in progress. On any project there are processes that are clearly working well and others that are not yet satisfactory. One of the difficulties of software engineering or software project management is to take the realities of a given project and mold them into a form that is in agreement with the theories of accepted development models. After realizing that a wholesale overhaul of the entire project to bring it into compliance with an accepted model was infeasible, but also that long-term use of sub-optimal processes was unacceptable, the Trilinos Project adopted a model of process improvement by which the processes that drive the project are always subject to ongoing, incremental revision and improvement. We always seek new processes or those modifications to existing processes that are likely to yield the most benefit at the least cost. The principle of process improvement is similar in spirit to the principle of iterative development, but while iterative development involves primarily the incremental improvement of the software, process improvement is the incremental improvement of the processes by which that software is developed.

4 Development Practices and Tools

The goals of the project and the principles that steer us toward those goals have been outlined. Next, we discuss the practices and tools Trilinos uses that, guided by the aforementioned principles, help us reach our goals. The development practices and tools listed here address many different

software development issues and have been carefully chosen to serve the needs of the project and to minimize overhead while producing the most benefit.

4.1 Source Management

Reliable source management is critical to a stable software development environment. Important features of a source management tool include providing backups and version control. Version control allows developers to revert to previous versions of the code with a simple command and also makes managing releases easier by providing the ability to create release branches that can be developed concurrently with the main development branch, while allowing one change to be applied to both branches. Having a centralized code repository is invaluable for multi-person development teams as it makes it easy to get the changes that others have made and provides an easy way to resolve differences in changes made by two people. It also allows close collaborators to get instant access to the absolute latest versions of the code, which reduces the length and complexity of iterations. A repository is also useful for storing files that control other tools (documentation, website, test harness); this enables convenient control in a centralized place, but with decentralized access.

Trilinos source code is maintained in a Concurrent Versions System [3] (CVS) repository. While there are now a number of source management tools with attractive features, CVS continues to meet our needs well. The cost associated with migrating to another source management tool and forcing all developers to learn a new system can not currently be justified by the small gain in features.

In addition to CVS, we also use Bonsai [6], a web-based interface to the information stored in the CVS repository. This allows developers to easily see changes made to source code, who made the changes, what log message they supplied, what code branch it happened on, and more. Bonsai has proved to be an invaluable supplement to CVS.

4.2 Communication Channels

The value of open lines of communication within a project cannot be overemphasized. Communicating requirements, design decisions, and timelines with all team members naturally promotes process improvement and leads to better, more efficiently developed code. Much of the electronic communication within the project is carried by email lists provided by a simple tool called Mailman [4].

Mailman list archives are searchable, which allows new Trilinos developers to catch up on interesting events from the past and stay up to date on current development without the risk of someone forgetting to CC them on an email. There are separate lists for user and developer conversations

as well as for announcements. CVS checkins and nightly test results are also sent to mail lists that developers can subscribe to. When committing changes to files in the CVS repository, developers are prompted to supply a message describing the change. These log messages are included in the email and are available in the CVS repository either on the command line or via the online Bonsai interface.

4.3 Requirements and Issue-Tracking

An important step in achieving a high level of software quality is tracking enhancement requests and issues pertaining to faults in the software. The Trilinos team uses a tool called Bugzilla [7] to automate this process. The interface for entering and searching for bugs is web-based, user friendly, and customizable. Dependency-tracking features simplify the task of tracking the relationship between bugs. The Trilinos team uses the concept of a metabug, which is a larger task that is dependent on multiple smaller tasks. Metabugs make it is easy for project leaders (or management) to track the status of issues that depend on many smaller tasks that are to be completed by one or more team members.

Although tracking issues in this way does not help to complete the necessary tasks any quicker, it does allow tasks to be properly prioritized, makes sure that issues are not lost or forgotten, and allows project leaders to quickly summarize the current state of the project.

4.4 Documentation

While the means, style, and content of documentation can be hotly debated topics, few will argue the need for some form of good source documentation. In a project of any non-trivial size, merely having comments within the source is insufficient. It becomes too inefficient to sift through thousands of lines of source by hand just to find what arguments a function takes. In Trilinos, we have adopted the use of Doxygen [9]. Doxygen allows developers to maintain documentation inline, but then parses the source files and generates browsable output in a number of formats. The Trilinos framework has taken it a step farther and set up mechanisms by which documentation is automatically generated twice daily from the latest versions of the source code and posted online. Having this documentation up-to-date and readily accessible online helps to improve interoperability and maximize the amount of support users can access themselves without needing to contact the development team.

4.5 Configuration Management

Achieving a high level of software quality is complicated when a software project consisting of many largely-

autonomous components needs to run on a wide range of platforms. The current Trilinos build system is based on GNU Autoconf [1] and Automake [2] [10], which help to minimize the amount of work needed to build the software on many platforms.

As mentioned in section 3.2, New Package can be used to quickly set up an Autoconf- and Automake-based build system for a new or existing piece of software. No current tools make configuration management a trivial issue; however, a configure and build system using Autoconf and Automake has been a noticeable improvement over a more traditional system using simple makefiles.

4.6 Information Distribution

In any complex software project, there is inevitably a lot of information that needs to be transferred from the various creators of this information to the consumers of it. This includes everything from contact information, documents, publications, presentations, bug reports, and frequently asked questions to the software itself. The natural choice for the delivery of all this information is a project website. It might seem painfully obvious that this is a good solution for a project's information distribution needs, but it is woefully underutilized by many CSE software projects. Like all of these tools, a project website need not be perfect and polished; it just needs to serve its purpose. So much of the value of a project website, whether it is for the development team only or for the general public, can be had with a very small time investment and a beginner's knowledge of HTML. As the website grows incrementally, the growing pains can be greatly alleviated with a little bit of CSS [12] and PHP [8].

One of the greatest benefits to be had from a project website is the ability to bring together the rest of the project's tools. If only a very simple list of links, having a comprehensive starting point from which to reach all of a project's resources is invaluable.

4.7 Testing

The success of any software project is critically dependent on good testing. Testing can be a painful activity when there is no good system in place to support it. Like any other activity, if it has to be done manually and from scratch every time, it will be prone to errors and it will not happen as often as it should. To address this, Trilinos has developed over the years a suite of scripts to run all tests on a number of different platforms automatically on a regular schedule. This system includes a standard interface for adding new tests, which then get automatically included in the testing. This helps to lower the barrier for developers to write and maintain valuable tests.

With a project the size of Trilinos, in addition to the testing itself, the collection, organization, and distribution of results are particularly challenging tasks. To address this, we have developed a database for results, which is then queried to display the latest results on the website. Summary emails are also generated and sent out each morning. This way, no critical bug should live for more than 24 hours without being detected. Providing good information to developers about the state of the code across all target platforms every day goes a long way to improving quality by tightening iterations.

4.8 Release Process

The Trilinos Project has invested a lot of time into improving its release process. We have established a release process timeline to ensure that all release activities are accomplished on time. Release process checklists are completed at the framework and package levels for each release, and checklists, along with associated issues, are stored in Bugzilla. All appropriate dependencies are tracked. The release candidate code is subjected to tests on each of the Trilinos nightly test harness platforms, as well as the acceptance tests of some of our most important customers. The timeline and checklists, along with the structure provided by Bugzilla, have been key in organizing the complicated efforts of a large number of developers in such a way that releases can be provided on time, and with confidence in the code.

The release process has benefited greatly from incremental process improvement. The initial timeline and process checklists were created based on the what worked fairly well in the past. By guaranteeing that important steps would be completed for future releases and making the effort to improve the processes after each release, there has been noticeable improvement to the release process after every Trilinos release cycle.

5 Conclusion

Historically, software quality assurance and related software engineering processes and concepts have not been a primary focus for CSE software projects. Furthermore, standard software engineering approaches used for business applications cannot be naively applied. At the same time, as CSE applications become increasingly part of high-risk, predictive decision-making, SQA processes and tools will be necessary.

Developing quality CSE software is challenging. Finding the time, energy, and resources to improve the processes by which you develop it can be even more so. Often the biggest obstacle is the mere thought of the daunting task of getting from where you are to where you want to be.

But, through organic, just-in-time adoption of these simple, proven, freely-available tools and techniques, one can incrementally improve the quality of a project's processes which will, in turn, improve the quality of the software. This approach has been very successful for the Trilinos project and appears to be appropriate for other projects as well.

6 ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the ASC and LDRD programs that funded development of Trilinos and recognize all of our fellow Trilinos contributors: Teri Barth, Ross Bartlett, Paul Boggs, Erik Boman, Todd Coffey, Jason Cross, David Day, Clark Dohrmann, Michael Gee, Robert Heaphy, Ulrich Hetmaniuk, Robert Hoekstra, Russell Hooper, Vicki Howle, Jonathan Hu, Tammy Kolda, Kris Kampshoff, Sarah Knepper, Joe Kotulski, Richard Lehoucq, Kevin Long, Joe Outzen, Roger Pawlowski, Eric Phipps, Andrew Rothfuss, Marzio Sala, Andrew Salinger, Paul Sery, Paul Sexton, Ken Stanley, Heidi Thornquist, Ray Tuminaro and Alan Williams.

References

- [1] Free Software Foundation. Autoconf Home Page. <http://www.gnu.org/software/autoconf>, 2004.
- [2] Free Software Foundation. Automake Home Page. <http://www.gnu.org/software/automake>, 2004.
- [3] Free Software Foundation. Gnu CVS Home Page. <http://www.gnu.org/software/cvs>, 2004.
- [4] Free Software Foundation. Gnu mailman home page. <http://www.gnu.org/software/mailman/mailman.html>, 2004.
- [5] M. A. Heroux. Trilinos home page. <http://software.sandia.gov/trilinos>, 2004.
- [6] The Mozilla Organization. Mozilla Bonsai Home Page. <http://www.mozilla.org/bonsai.html>, 2004.
- [7] The Mozilla Organization. Mozilla Bugzilla Home Page. <http://www.mozilla.org/projects/bugzilla>, 2004.
- [8] The PHP Group. PHP Home Page. <http://www.php.net/>, 2005.
- [9] D. van Heesch. Doxygen home page. <http://www.doxygen.org>, 2004.
- [10] G. Vaughan, B. Elliston, T. Tromeu, and I. Taylor. *Gnu Autoconf, Automake, and Libtool*. New Riders, 2000.
- [11] D. Wells. Extreme Programming: A Gentle Introduction. <http://www.extremeprogramming.org>, June 2006.
- [12] World Wide Web Consortium. Cascading Style Sheets Home Page. <http://www.w3.org/Style/CSS/>, 2005.