Software Engineering Best Practices  Andy Salinger, SNL

The overall goal of the Software Productivity for Extreme-Scale Science effort is to improve the productivity of our computational science community to develop predictive simulation codes for DOE missions. For this goal, the codes must be efficiently developed and maintained, be trusted in their predictions, and perform well on current and future architectures. Software Engineering is the discipline that facilitates these characteristics.

As we tackle more complex applications, DOE codes must increasingly be built on diverse sets of expertise, with large interdisciplinary teams, and making use of codes and libraries developed external to the project. The era of the Hero Programmer is over. The challenge is to maintain productivity as projects grow in number of developers, geographic separation, diversity of background and roles, size of software, number of platforms to support, and number of programming languages. For productivity, the software development process itself (not just the algorithms) must be scalable. In this case, scalability is with respect to the number of developers and code features, not problem size and processor count.

To achieve scalability in code development for DOE scientific simulation projects, we need to increase investment in the use of software quality tools and processes, and to evolve the culture to one that recognizes, and reaps, the value of software engineering. The current state of software engineering among Office of Science application projects varies widely. The variation is in part a cultural phenomenon, but also will (justifiably) be a function of where the project fits on the spectrum of fundamental research to decision support and on the project size and geographic distribution of developers.

We propose to assemble a “Software Engineering Best Practices” reference for DOE application projects. We will draw from the experiences of the computational science efforts that have been successful in achieving scalability of development, support, and maintenance for large-scale projects. These projects have in turn drawn from the broader software engineering literature. What we propose here is a distilled reference targeted for the DOE scientific application community. See Appendix A for a draft of the tools and processes, and Appendix C for a list of cultural principles, that a project could consider adopting. Since projects vary in culture, size, and software engineering needs (and because changes in software tools can only be tolerated in small doses), the suggested Best Practices need to be presented with multiple levels of formality for incremental adoption. See Appendix B for a prototype of a checklist format.

We propose to assemble a “Software Engineering Best Practices” reference for DOE application projects. We will draw from the experiences of the computational science efforts that have been successful in achieving scalability of development, support, and maintenance for large-scale projects. These projects have in turn drawn from the broader software engineering literature. What we propose here is a distilled reference targeted for the DOE scientific application community. See Appendix A for a draft of the tools and processes, and Appendix C for a list of cultural principles, that a project could consider adopting. Since projects vary in culture, size, and software engineering needs (and because changes in software tools can only be tolerated in small doses), the suggested Best Practices need to be presented with multiple levels of formality for incremental adoption. See Appendix B for a prototype of a checklist format.

The Software Engineering Best Practices list is not meant to be a set of requirements for projects. It is meant as a reference where computational science projects can get advice and more detailed references on what software engineering practices the broader community recommends. The following appendices could provide a starting point for this effort.

Appendix A: Best Practices in Software Quality Tools and Processes
A number of software quality tools and processes are emerging as “best practices” in the computational science community. This draft list would be the starting point for the more complete, and layered, product of this effort. Each entry in the best practices list would be presented with multiple levels of formality (see Appendix B).
1. Automated backups to store code and data
2. Source code repository with version control (e.g., SVN, git)
3. Build system for compilation and linking of code
4. Configuration management (e.g., CMake, autoconf) for selecting code features and setting machine-specific options and paths
5. Mailing lists for developers, users, and automated test results
6. Verification problems that assess correct coding of both equations and algorithms, with analytic/manufactured solutions and benchmark problems
7. Regression test in an automated test harness with simple pass/fail result, for developers to use before committing code
8. Automated “nightly” testing of code against with coverage of common configurations and platforms, with a web dashboard for navigating results
9. Automated “nightly” continuous integration against key Third-Party Libraries (TPLs)
10. Unit test framework and pervasive unit tests for more thorough testing, more rapid bug determination, and more modular software design
11. Code coverage tool for assessing test coverage
12. Use an IDE for efficient navigation of large code bases (e.g. Eclipse)
13. Example problems that demonstrate breadth of capabilities
14. Independent testing for performance, scalability, and memory leaks
15. Theory Manual that shows what equations are being solved, with verification problems
16. Developer’s Guide for new developers to get up to speed rapidly
17. User’s Guide documenting the interface and input files
18. Release processes including licensing and Backward Compatibility policy
19. Agile coding practices: e.g. rapid code investigations, test-driven development, frequent code refactors, frequent re-prioritizations of tasks, pair programming, code reviews.

Appendix B: Best Practices in Software Quality Tools and Processes
Each entry in the Best Practices list will be expanded to provide a checklist of tools and process that a project can consider adopting. Each entry will have a link to provide more details.

2. Source Code Repository with Version Control
   - Store code in version controlled repository (SVN or git) with access controls (details)
   - Automate commit messages to mailing list of developers (details)
   - Establish branch conventions: naming, merge frequency, release tags, bug fixes (details)
   - Capture provenance: capture version tag for output in a log file (details)
   - Checkin test script: accept commits only when a set of screening tests pass (details)
   - Code review tool: only accept commits after code review (details)

Appendix C: Best Practices for a Software Engineering Culture
In addition to Software Tools and Processes above, there are elements of a software engineering culture that aid in maintaining productivity as we evolve to larger, more distributed, more complex, more mission-oriented software projects. Some elements of a software engineering culture could include these tenets:

1. All code that you care about must be tested in the automated regression suite so others can develop code and commit changes with confidence.
2. Don’t break the trunk: for everyone to remain productive, the regression test suite on the development platform should remain on the manifold of 100% Tests Passed.
3. Fix broken code quickly: commits that cause tests to fail, perhaps for configurations and platforms other then the standard development environment, or for scalability or performance, must be fixed right away or backed out of the version control code.
4. Write modular code, such as object oriented: modular code has numerous benefits for flexibility and extensibility. For one example, part-time and new developers can make an impact by just understanding a limited section of the code.
5. Have a Code Czar in addition to the scientific PI. The skills and temperament needed for these two roles often mean that different team members hold these titles.
6. Do code reviews. Before committing substantial changes, get another developer to assess your changes for software design, efficiency, clarity, redundancy, and extensibility.