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

Michael A. Heroux
James M. Willenbring
Sandia National Laboratories

• Special Thanks:
  • LDRD
  • NNSA ASC
SAND#: 2009-0579 C
Target Audience:
Research CSE Software

- Typically developed using research funding.
- Formal software engineering seldom a primary goal.
- Research CSE software developers:
  - Often lack the training, resources or time to adopt advanced formal methods and practices
  - Have a skeptical view of formal software engineering practices.
- Our theme: Better SE ➔ Better Research CSE
  - Select only those practices that we are confident can pay off.
  - Introduce them gradually.
The 10 Practices

- Identified from Trilinos project.
- Focus: Practices that most research CSE software teams can adopt and benefit from.
- Similar to Agile processes.
- Additional practices are valuable, but:
  - Heavy emphasis on SE can be a distraction.
  - Practices must be introduced gradually.
Research vs. commercial software

- **Commercial software**
  - Primary purpose: generating revenue
  - Domains: Underlying algorithms and methodologies are mature
  - Increasingly sophisticated and complex, yet more easily developed and more reliable.
  - Reason: software engineering is more mature.

- **Research software**
  - Primary purpose: Generating science results.
  - New algorithms and modeling capabilities.
  - Software developed as proof-of-concept and to generate first-of-a-kind results
  - Highly trained scientists, not professional software engineers.
  - Scientists can produce high quality software:
    - Use common sense principles and self-discipline.
    - But ad hoc manner makes it difficult to leverage a product outside its narrowly intended scope.
Practice 0: Manage source (the basics)

- The vast majority of CSE software projects use source management
- But not all.
- Single most important practice:
  - Source files are kept in a repository.
  - Developers regularly commit changes.
  - Repository is *the* source for source code.
Practice 1: Use issue-tracking software for requirements, features and bugs

- Issue-tracking software:
  - logical collection point for information concerning bugs, features, and requirements.

- Why:
  - Issues visible to the whole team
  - Ability to prioritize issues
  - Ability to establish dependencies between issues:
    - Break larger issues down into pieces.
    - See how different issues affect one another
  - History of issues searchable.
Practice 2: Manage source (beyond the basics)

- **Branching**
  - Independent line of development (not agile).
  - Stabilize a release branch (not agile).
  - Still can merge from one branch to another (challenging).

- **Tagging**
  - Snapshot of the current state of the repository.
  - Create a bit-wise identifiable release.
  - Eliminates ambiguity.

- **Source browsing and viewing tools**
  - ViewVC - can be used with SVN or CVS
  - Bonsai - compatible only with CVS
    - Search.
    - Browse.
    - Compare.
Practice 3: Use mail lists to communicate

- Abstraction of interested people.
- Centralized mail list tool prevents the lists from getting stale.
- Useful for archival purposes and spam filtering.
- Examples lists:
  - Users
  - Developers
  - Leaders
  - Regression
  - Check-in
  - Announce
- Wikis may be used in addition to mail lists, advantages:
  - hypertext browsing
  - real-time editing
  - collaborative development of content.
Practice 4: Use checklists for repeated processes

- Checklists are valuable tools
  - making easily repeatable processes
  - reduces the chance that steps are omitted
  - training purposes
  - artifacts

- The Trilinos project uses several different checklists
  - several release checklists
  - a new developer checklist
  - a CVS commit checklist

- Goal: Automating checklist steps is even better.
About “Barely sufficient”

- A minimalist attitude to formal processes:
  - Adopt only those that have a large impact.
- Mindless Imposition of Formal SE bad for CSE community:
  - Large-scale formal document generation as “first step”.
  - Large effort to satisfy an external requirement, does not benefit the project team.
  - Documents become out-of-date quickly and therefore are irrelevant or even misleading.

- Formal documents:
  - Certainly play a role in a project:
    - Domain vision statement, e.g., Trilinos Strategic Goals.
    - Highlighted core, ACM TOMS article *An Overview of the Trilinos Project*.
  - Modest, should be developed after the product architecture is stable.
  - Are essential when a product is ready for hand-off to maintenance team.
Practice 5: Create barely sufficient, source-centric documentation

- Create a combination of near-to-the-source and in-source documentation can be very effective.

- **In source:**
  - User-callable functions and executables should be documented in the source files, e.g. using Doxygen.
  - Processing source files then generates documentation.

- **Higher-level conceptual documentation:**
  - Custom-developed, but still tightly coupled to examples in software repository.
  - Examples should be extracted from actual working examples in the repository.

- **Requirements, analysis and design documentation:**
  - Captured by appropriate tools such as Bugzilla (for requirements) and UML graphics tools (e.g., Microsoft Visio).
  - Doxygen for design discussions: produces UML diagrams directly from source code.

- **Documentation efforts should not:**
  - Be long, hand-written, text documents until …
  - a project reaches a level of maturity where there is little change in software design and implementation.
Practice 6: Use build-configuration management tools

- **Build-configuration management tools:**
  - Makes software accessible to a much broader audience.
  - Make software support much less expensive.
  - Superior to hand-written makefiles (common for CSE software).

- **Preferred approaches:**
  - Linux RPM or Windows.

- **About Cmake:**
  - Very portable, supplies rich set of build targets.
  - Trivial to use for simple projects.
  - Complex codes:
    - Configuration management tools challenging to adopt.
    - Provide tremendous value in long run.
Practice 7: Write tests first, run them often

- Common view: Testing done late in coding process.
- TDD: Write tests first.

- TDD benefits:
  - Test programs debug design.
  - Measure of progress: 100% test failure to 100% test success.

- Full suite of tests provides:
  - Confidence to revise after the initial implementation
  - Improves long-term quality of product as it matures.

- Adopting TDD as a habit:
  - A cultural challenge: writing tests delays the initial development.
  - But provides tremendous value: Greatly reduces development costs, improves long-term software quality.
Practice 8: Program tough stuff together

- **Pair programming:**
  - Concept formalized by XP.
  - Not natural for CSE developers.
  - More used to sitting alone carefully writing source.

- **Selective use:**
  - Don’t advocate pair programming for all development.
  - Development of complex software functions.

- **Especially:**
  - Incorporating the use of another developer’s software.
  - Produces superior software.
  - Provides important feedback.
Practice 9: Use a formal release process

- **When a project is just getting started:**
  - Run some reasonable set of tests on a defined set of platforms
  - Tag the new version when all of those tests pass.

- **For larger software projects:**
  - Formal release process is essential.
  - 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.

- **Continual Process Improvement (See Practice 10):**
  - Trilinos and its user base have grown dramatically.
  - Release process for a major release has gone from an informal series of tests on a release branch to a much larger, coordinated effort.
  - 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.

- **Minor releases:**
  - Entire major release process no justified. the cost.
  - Subset of the major release process is used.
  - Periodically evaluated for effectiveness.
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.

- Example: Training a new developer.
- Until a draft process is recorded, user training will be haphazard.
- Standardize the training with a checklist.
- Refine checklist using process improvement.

- Checklist usage:
  - Each use: consider whether or not modifications are necessary.
  - Poll checklist users to combine all of the best ideas into one standard list.
  - Include items on process checklists that reflect future goals
Conclusions

- Research CSE software can benefit from modern software engineering practices and processes.
- However:
  - The goal of research CSE software is often research and development.
  - The software product is just one output.
  - Too much emphasis on software processes can put a project at risk.
- The 10 practices: Not a large effort for most research CSE software teams.
- Once adopted should:
  - Provide a qualitative improvement in the overall software development process, producing better quality software with less effort.
  - Give research CSE project teams more time for science and engineering research and development.
References