Bringing Best Practices to a Long-Lived Production Code

Charles R. Ferenbaugh

HPC Best Practices Webinar
January 17, 2018

National Nuclear Security Administration
Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

Disclaimer

This talk comes in two parts, a general philosophy part and a case study part

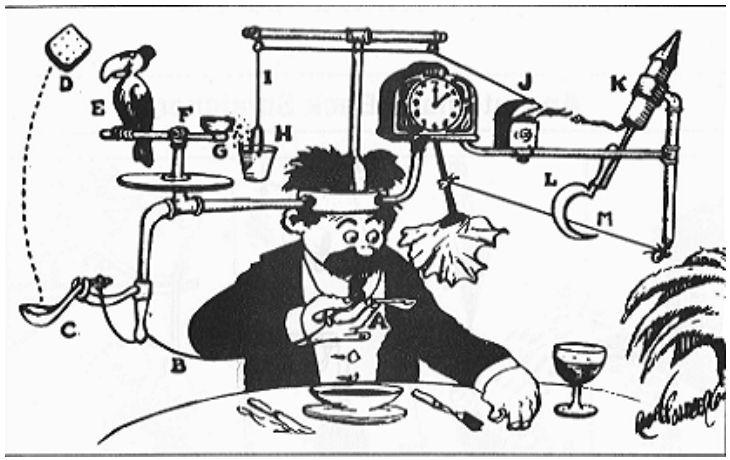
- The general part applies to many (all?) long-running scientific software projects
- The solutions from our case study may or may not apply to your project; they're meant as examples

Outline

- Problems faced by long-lived scientific codes
- LANL's experience in the xRage code project
- Recommendations for other projects

Long-lived scientific codes

- Discussions of best software practices sometimes assume (implicitly?) that you're starting a new project and a new code
- But what if you have an ongoing, years- or decades-old project?
  - Large, pre-existing code base
  - Existing code team with established habits
  - Significant user base, already using the code regularly
- Often such projects have major challenges to software quality
  - Complex, hastily-written code
  - Incomplete testing
  - Inadequate documentation
  - Little or no software process
  - A culture that says, "Why should we do all this fancy process stuff? We're getting along fine without it!"



What do you mean by “getting along fine”?

- Historically, it has usually meant that the code:
  - Has the capabilities the users want
  - And has them ASAP
- This approach can be successful in the short term. . .
  - Can build up a user base
  - Can meet deliverables, produce papers, get grants renewed, etc.
- . . . but it has problems that show up in the longer term
  - Code is written hastily, hard to understand
  - Design is ad-hoc
  - Difficult for code team to maintain, extend
  - Difficult for new team members to learn
  - Difficult to optimize for new architectures
- In other words, it’s not sustainable

What do you mean by “getting along fine”? (2)

- A modern, better definition would be that the code:
  - Is understandable, maintainable
  - Is extensible
  - Is well-tested
  - Is well-documented
  - Is portable to modern architectures
  - . . . And still has the capabilities the users want
  - . . . And has them (reasonably) quickly
- This is more sustainable for the long term

Changing practices requires changing values and culture

- A project decides what it values, and grows a culture that reflects those values
- This affects many aspects of a code project:
  - Languages, programming models, tools used (or not used)
  - Staffing (how many developers? what background?)
  - Training, career development
  - Performance evaluations
  - Tasking, scheduling, deliverables
- These all reinforce each other, push the project in a certain direction
- It's very hard to change that direction without (at least partly) changing values and culture

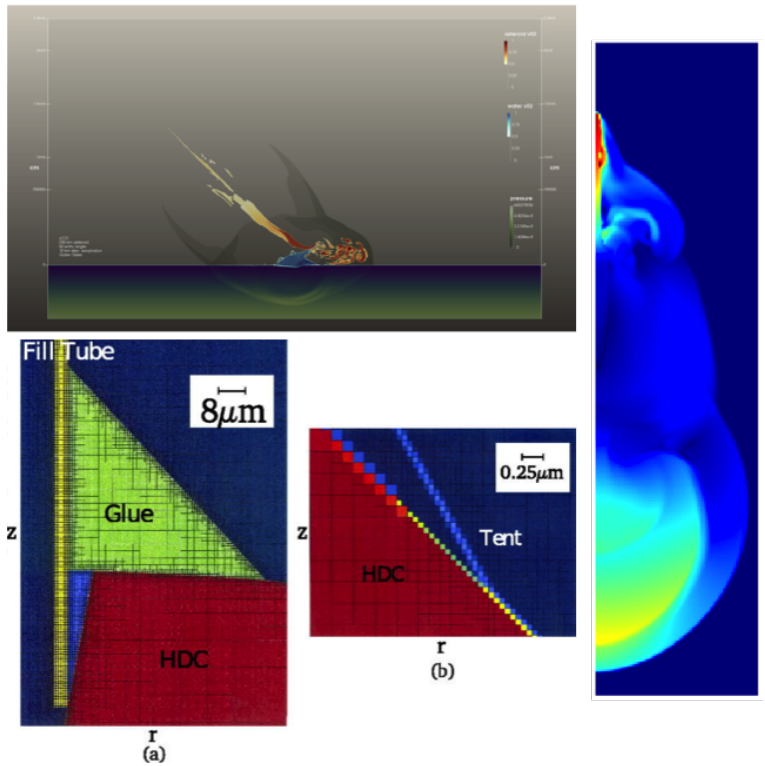
Changing practices can require changing code

- Sometimes best practices and modern tools have built-in assumptions that older codes don't satisfy:
  - Unit testing assumes self-contained units
  - Shared ownership of code assumes understandable code that any developer can reason about
  - And so on...
- Result: changing practices may have to go hand-in-hand with changing code
  - This may make starting the process harder
  - But once it does start, it can become a "virtuous cycle"

So what does it look like to put all this into practice?

Case study: The LANL xRage code

- xRage is an Eulerian AMR radiation-hydrodynamics code
- Original code written ∼1990
- Has been used successfully in several application areas
- Contains about 470K lines of source code
  - Not counting numerous third-party libraries, from LANL and elsewhere
- Mostly Fortran 90, some C/C++
- MPI-only parallelism



xRage applications: asteroid impact simulations, shape charge experiments, Inertial Confinement Fusion simulations

The need for modernizing xRage

20+ years of high-pressure work left xRage with significant technical debt. This made it difficult to:

- understand the code flow or data flow
- maintain the code
- add new features
- *train new developers as older staff retire*
- *refactor for advanced architectures, such as Trinity, Sierra, . . .*

These factors (especially the last two) made us realize that things needed to change!

Prerequisite #1: Management support for culture change

Management saw the need for doing things differently, was willing to make changes:

- Added a CS co-lead to the project
- Shifted project resources to support more CS/SE staff
- Allocated part of domain scientists' time to modernization work
- Scaled back development of new physics features, milestone commitments

Prerequisite #2: Regression test suite

- Before: We had a regression test suite, but it wasn't well-maintained
- As refactoring started: team committed to keeping tests passing ("wall of green")
- At first, all tests were integrated tests
  - Unit tests were added later
- Nightly, weekly test runs are automated, results emailed to team
- Tests serve as a safety net as we refactor

Collaborative Test System

Los Alamos National Laboratory Collaborative Test System

APPLIED PHYSICS DIVISION

CRESTONE Project
Tue Jul 26 07:00:43 MDT 2016

CROSS PLATFORM RESULTS (Nobel)			
	CI	ML	TR
	intel	intel	intel
	default	openmpi	default
	xrage	xrage	xrage
Totals	147/147	147/147	146/146
RunTime	13234	15270	6725
Test			
AMR_dart	Pass 239	Pass 189	Pass 109
AMR_dblast	Pass 187	Pass 235	Pass 111
AMR_sod	Pass 26	Pass 66	Pass 18
AnisoFailStretch	Pass 74	Pass 410	Pass 81
Cond	Pass 12	Pass 22	Pass 9
CrushAlum	Pass 99	Pass 110	Pass 56
CuAirRP	Pass 76	Pass 48	Pass 36
Exact_Cond1dLin	Pass 8	Pass 10	Pass 5
Exact_Cond2dLin	Pass 138	Pass 105	Pass 57
Exact_Cond2dNl	Pass 140	Pass 101	Pass 58
HE_AR	Pass 70	Pass 109	Pass 41
HE_CerroGrande	Pass 35	Pass 35	Pass 21
HE_FF	Pass 11	Pass 12	Pass 8
HE_FFPB	Pass 237	Pass 182	Pass 113
HE_FF_RS	Pass 23	Pass 35	Pass 18
HE_IG	Pass 22	Pass 28	Pass 15
HE_PB	Pass 15	Pass 30	Pass 13
HE_SURF	Pass 231	Pass 245	Pass 121
HE_SURFmerge	Pass 13	Pass 15	Pass 10
HE_SURFplus	Pass 203	Pass 191	Pass 110
HE_SURFplusZND	Pass 48	Pass 33	Pass 32

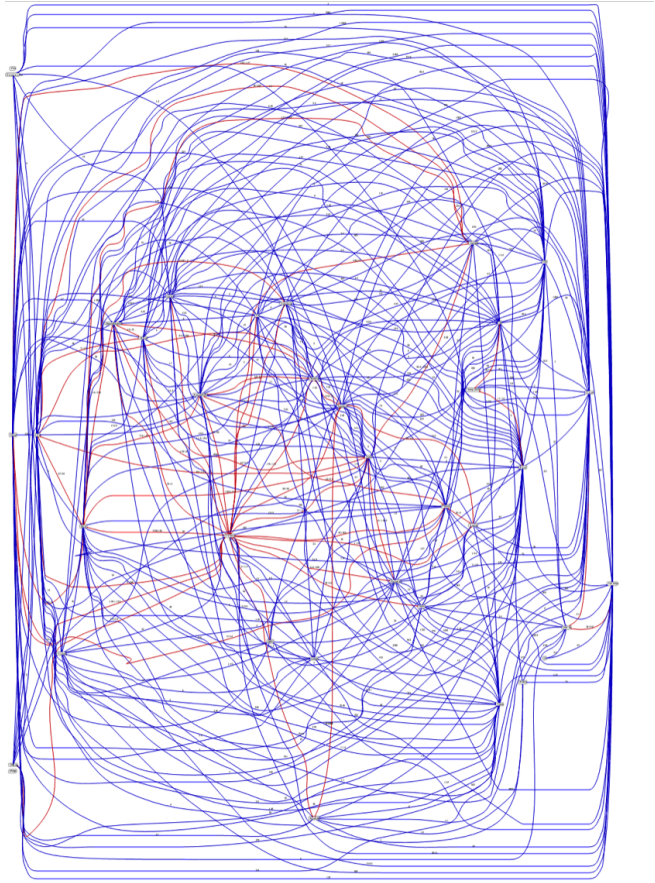
What to tackle first?

Several possible tasks:

- Move to modern build system (e.g. CMake)?
- Implement unit testing?
- Clean up our tangled dependency structure?

We decided to do cleanup first

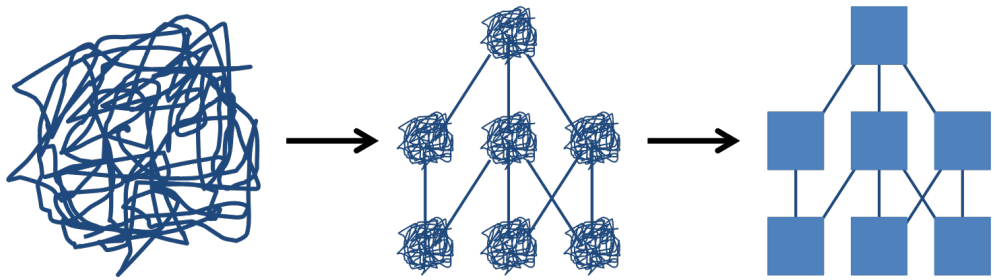
- Cleaner code has immediate benefit
- Can't do unit tests on a hairball code
- *Could* use CMake on a hairball code, but that's not what CMake is designed for



xRage dependency graph, 2014-10-01 (the "hairball" graph)

Untangling dependencies

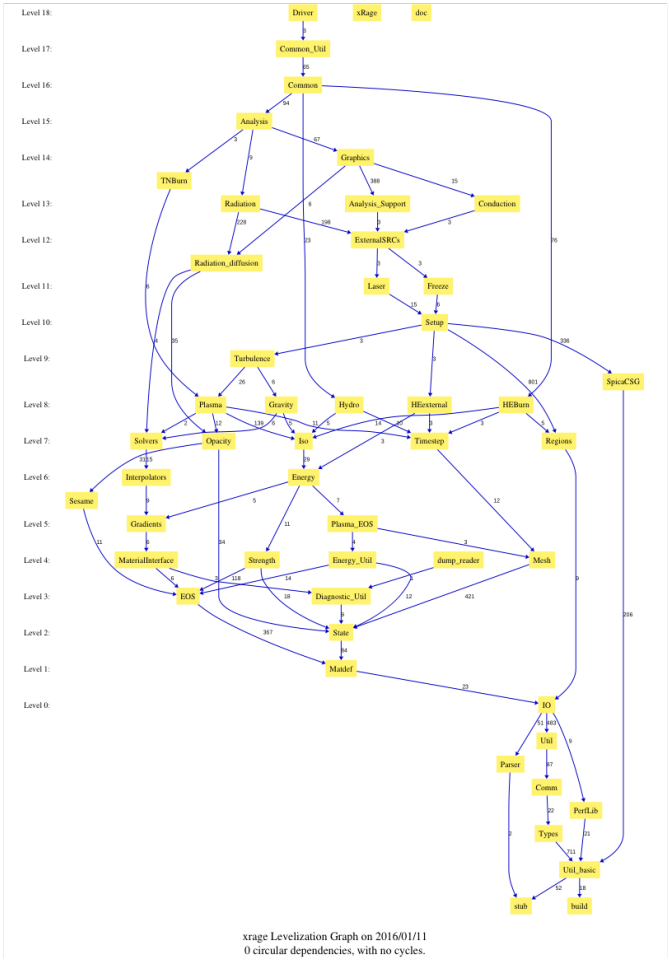
- Any file could use data, call routines from any other file
- Our strategy to change this:
  - Change existing code base *in place*
  - Separate code into *packages* of related functionality with well-defined interfaces
  - Move toward a cleaner, simpler design
- Some techniques:
  - Create derived types for package state, pass through argument lists
  - Find misplaced code and move it to a proper place
  - Lift some function calls (e.g., coupling) to higher-level packages
  - Deprecate/remove unneeded calls



Untangling dependencies (2)

After about 15 months of work, this process led to a much simpler graph (right)

- Graph is levelized, has no cycles!
- Interfaces between packages are better-defined
- This makes it easier to understand, reason about the code
- This enables other changes *on a per-package basis*
  - Unit testing, documentation
  - Code cleanup
  - Performance optimization
  - Physics improvements



xRage dependency graph, 2016-01-11

Where we are now

Task list:

- Levelize dependency graph (complete)
- Refactor build system to use libraries, enforce levelization (complete)
- Add unit tests (infrastructure complete, test writing ongoing)
- Document packages (ongoing)
- Clean up code within packages (ongoing)
- Work on performance optimization (ongoing)
- Move from home-grown build system to CMake (prototyped)
- Move from SVN version control to Git/Gitlab (planning)
- Set up Gitlab-CI continuous integration (planning)

Some recommendations to other projects

- Get management support for culture change - this is crucial!
- Use regression tests as a safety net as you refactor
- Resist the temptation to move to a shiny new tool just because it's shiny and new
  - Prioritize tasks/changes by value added to the project
- Find the right balance between code/process improvement and user support
  - Both are important!

Resources

General resources:

- Lakos, *Large-Scale C++ Software Design* (1996)
  - Specific mechanisms are now outdated, but...
  - General principles still apply to all languages, not just C++
- Feathers, *Working Effectively with Legacy Code*

More details on xRage refactoring:

- Ferenbaugh et al., *Modernizing a Long-Lived Production Physics Code*, SC16 poster
  `http://sc16.supercomputing.org/sc-archive/tech_poster/tech_poster_pages/post196.html`

Resources (2)

Tools we've found useful for xRage:

- **Understand** static visual analysis tool
  `http://scitools.com`
- **Graphviz** graph visualization for dependency graphs
  `http://graphviz.org`
- **pFUnit** unit test framework for Fortran
  `http://pfunit.sourceforge.net`
- **Google Test** unit test framework for C/C++
  `https://github.com/google/googletest`

Questions?

Thanks for your attention!

Charles Ferenbaugh
cferenba@lanl.gov