How Not to Write Software Libraries

William Gropp
www.cs.illinois.edu/~wgropp

Some Background

- *Why we couldn't use numerical libraries for PETSc*.
  - In Ronald F. Boisvert, editor, Proceedings of the IFIP TC2/WG2.5 Working Conference on the Quality of Numerical Software, Assessment and Enhancement, pages 249–254. Chapman & Hall, 1997.
- *Exploiting existing software in libraries: Successes, failures, and reasons why*.
  - In Michael Henderson, Christopher Anderson, and Stephen L. Lyons, editors, Object Oriented Methods for Interoperable Scientific and Engineering Computing, pages 21–29. SIAM, 1999.

- Much of this is unfortunately still true…

First, Why Write A Library?

- Promote a new algorithm, become famous
- Solve one problem
  - I.e., piece of my application

- Write what is needed to solve some problem**s**
- Whose problems?
  - Yours and ?

Who is the Customer?

- The customer cannot be everyone!
  - Failing to identify the customer is the first (but still fatal) step toward failure
- You should be one of the customers
  - Common failure mode: people who don't use their own product (you see this often in reviews – "didn't they try this?!!")

What Do They Want?

- Few customers want a particular algorithm. They want a *solution*.
- Tradeoffs
  - ♦ Convenience, simplicity, performance, correctness, robustness,…
  - ♦ BLAS (esp. levels 1 and 2 but even 3) use character strings to select from related operations (e.g., transpose an argument).
  - ♦ Tradeoff: Fewer routines *at the expense of more overhead, which increases the minimum size at which the routine performs faster than simple user code*

What Do They Need?

- Classic misunderstanding: "Invert a matrix"
  - **Need** to find an approximate solution of a linear (or more likely, non-linear) problem
- More recent (and far more damaging to computational science): "need POSIX I/O" when ***no one*** needs POSIX I/O *semantics*
  - Most applications only need simple single (parallel) program read or write (not read **and** write)
  - A few need some sort of relaxed consistency model

Five Ways to Fail

1. Nonportable code
   - Unnecessary use of language extensions, invalid assumptions about datatype size (int is not 32 bits). Namespace pollution and poorly defined header files
2. Parallel code written for all processes only (COMM_WORLD in MPI)
   - MPI libraries that don't use a private communication context
3. Obscure or inappropriate data structures
   - From the *application's* view.
     - Block-cyclic may make sense for the algorithm writer but not for the application developers
     - Even banded format is weird for users

Five Ways to Fail

4. Slavish object-oriented design at the expense of performance
   - Closely related: Assumptions that the compilers can produce fast code (faster than *any* programmer)
   - We *know* this is not true, especially for vectorization
   - Also related – "warning-free compiles" even when the warning is incorrect
5. Global state, lack of modularity and assumptions about usage model
   - E.g., an FFT library that computes state on the first call and reuses that on subsequent calls – good if all FFTs are the same size; disaster if they alternate between two sizes (not a hypothetical case ☹).

Comments on Five Ways to Fail

- Still true 20 years later
  - Improvement in some areas, little in others
  - Compilers are much better, but still far from optimal (and vectorization just makes the situation worse)
  - Slavish object oriented design has become a chronic problem in computer science (see the ACA website disaster)

Five Ways to (improve the chances that you) Succeed

1. Respond to questions and bug reports
2. Provide documentation and examples
3. Pay attention to performance
   - And know what good performance is; don't assume that you and your compiler will provide it because you followed some rules
4. Don't confuse orthogonality of concepts with orthogonality of interface
5. Pay attention to the learning curve
   - Tutorials, "bring your own code" workshops, books

Four Issues to Remember in Building Components

- Portability
  - Pick a standard an enforce it. For C/C++, that is one of the ISO standards, not extensions (such as GNU), no matter how useful
  - Exception: If there is a *significant* impact on functionality or performance, *and* there is a fallback, make the use of extensions *possible.* Atomic memory operations are one such example; some vectorization extensions are another
- Avoidance of Global State
  - Harder than it sounds, and unavoidable for some (I/O to stderr, for example)
- Interoperability and Composibility
- Documentation, Examples, and Support
  - See above

Ten Mistakes Still Being Made

1. Ignorance of standards
2. Requirement to be the master
3. Printing error messages and/or exiting from the program
4. Makefiles for a particular system
5. No (or very poor) documentation
6. No testing
7. No examples
8. Name space pollution
9. Algorithm-oriented library
10. Requiring that all processors/cores/what have you be used

Ten Mistakes Still Being Made

- Ignorance of standards
  - Compounded by sloppiness about version (C99? C11? Fortran 2008? Fortran 2008 + unofficial but "blessed" extension for MPI?)
  - No excuse; most compilers do a good if not perfect job at flagging invalid statements
- Requirement to be the master (i.e., in charge)
  - There can only be one master. If you insist on being it, you better be prepared to do *everything*

Ten Mistakes Still Being Made

- Printing error messages and/or exiting from the program
  - Nice as an option, fatal as a requirement
- Makefiles for a particular system
  - How can this still be happening?
  - Icky build systems are no excuse. Live with it

Ten Mistakes Still Being Made

- No (or very poor) documentation
  - ♦ Documentation and examples are essential
    - This is like writing a paper. Producing software without documenting it is like proving a theorem without writing the paper explaining the result. You won't and **should not** get any credit without the documentation.
  - ♦ Automatic tools do not solve this. I've seen doxygen generated so-called documentation that was nearly useless (or maybe worse than useless because it pretended to be useful). That's not doxygen's fault – it's the fault of the developers for trying to avoid writing documentation.

Ten Mistakes Still Being Made

- No testing
  - Like makefiles, how can we not have learned? But still true far too often. Everyone should *require* that software, including all open source software, publish at least their coverage analysis, on a line-by-line basis.
- No examples
  - Really? And the equivalent of "hello world" doesn't count

Ten Mistakes Still Being Made

- Name space pollution
  - Unix sets a terrible example here. Do *not* make the same mistake
    - True story. Scientist wrote code involving binding energies, and used "bind" as a routine name. But bind is an obscure but critical network function in Unix (man section 2), causing strange failures in the parallel program.
  - Modern languages addressing this, but middleware developers still often sloppy
    - You can use nm to look at the symbols in your library. *Everything* you define should be easily identified and the namespace easily described. This check can (and has been automated). Everyone should insist that a report listing all global symbols be published.

Ten Mistakes Still Being Made

- Algorithm-oriented library
  - Remember: Algorithms + Data Structures = Programs, and many modern problems require nontrivial data structures
  - The library will need to fit into a larger context. How hard have you made that on the user by making it easy for the library developer?
- Requiring that all processors/cores/what have you be used
  - Still a problem with some parallel programming models (though most are trying to define teams)
  - Still open problem: Negotiating resources between components or programming systems

Common Themes

- Much can be done with automation
  - ♦ Compiler to check standard
  - ♦ Symbol name checks
  - ♦ Coverage analysis to check test coverage
  - ♦ Code style conformance
    - All styles are compromises. Don't argue about the style, just pick one and use it.
  - ♦ Documentation generation to handle mechanics of docs
    - But documentation, like code, still needs to be written (and rewarded)
  - ♦ Autotuning and code generation tools for performance

Common Themes

- Much can be done by insisting on openness
  - More than just open source code
  - Publication of code quality measurements, details of testing, code style conformance and symbol name checks
  - Open buglists, issues
  - "xfail" in tests *must* be reported
    - I.e., 10 tests would have failed if we had been honest enough to run them

Conclusion

- Writing good software is *hard*
- Let other people do it as much as possible
- If you do it, *take pride in it*
  - Use tools to help you do it better
  - Exploit the community to get feedback, ideas, embarrassment
  - Writing the code is the *easy* part
    - Testing, documentation, tutorials, papers, collaborations, …