Software Development Basics

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

A Roadmap to the Workshop

- Focus on software development concepts
- Introduce tools and processes for organizing development and maintenance
- Discuss strategies and best practices
- Explore methodology that encourages collaborative software development
- Favor writing reusable software frameworks
- Work in groups with complementary expertise

Conventional Software Development Process

- Start with set of requirements defined by customer (or management):

    features, properties, boundary conditions

- Typical Strategy:
    - Decide on overall approach on implementation
    - Translate requirements into individual subtasks
    - Use project management methodology to enforce timeline for implementation, validation and delivery
- Close project when requirements are met

What is Different in the Scientific Software Development Process?

- Requirements often are not that well defined
- Floating-point math limitations and the chaotic nature of some solutions complicate validation
- An application may only be needed once
- Few scientists are programmers (or managers)
- Often projects are implemented by students (inexperienced in science and programming)
- Correctness of results is a primary concern, less so the quality of the implementation

Why Worry About This Now?

- Computers become more powerful all the time and more complex problems can be addressed
- Use of computational tools becomes common among non-developers and non-theorists
  -> many users could not implement the whole applications that they are using by themselves
- Current hardware trends (SIMD, NUMA, GPU) make writing efficient software complicated
- Solving complex problems requires combining expertise from multiple domains or disciplines

Ways to Move Forward

- Write more modular, more reusable software
  => build frameworks and libraries
- Write software that can be modified on an abstract level or where components can be combined without having to recompile
  => combine scripting with compiled code
- Write software where all components are continuously (re-)tested and (re-)validated
- Write software where consistent documentation is integral part of the development process

In One Sentence...

Scientific software development has to be recognized as a task requiring trained specialists and dedication of time and resources to produce dependable results

Embedded Scripting Language

- Not a new idea, but many scientific tools with scripting have their own “language”
  -> script capability added on top of the tool
- Better to add domain specific extensions to an existing, generic scripting language:
  -> use a language designed for scripting
  -> can import other extensions, if needed
  -> better documentation for script language
  -> users may already know the syntax
- We will use Python in this workshop

Script Language Benefits

- Portability
  - Script code does not need to be recompiled
  - Platform abstraction is part of script library
- Flexibility
  - Script code can be adapted much easier
  - Data model makes combining multiple extensions easy
- Convenience
  - Script languages have powerful and convenient facilities for pre- and post-processing of data
  - Only time critical parts in compiled language

Modular Programming & Libraries

- Many tasks in scientific computing are similar
  - Tasks differ only in some subset of the calculation
  - Calculations use common operations like fast Fourier transforms, basic linear algebra, etc.
  - Data can be represented in a structured file format supported by generic analysis & visualization tools
- There is a large potential for code reuse
- Independent modules can be better validated
- Reusable code is better target for optimization

Object Oriented Programming

- Provide levels of abstraction
  -> no need to know how something is done
  -> opportunity to transparently optimize (for platforms, if certain conditions are given, etc.)
- Organize access to data
  -> combine data with functions that modify it
  -> control read-only vs. read-write access
  -> handle side effects, on-demand computation
- Preserve APIs and favor local changes
  -> modifying one part does not break others

Unit and Regression Testing

- Complex software cannot be fully tested, but
  - Many components can be tested individually
  - Testing of individual units is fast, can be automated
  - When testing individual units, you can also test for the correct handling of incorrect use or data
  - Failures in individual units may not always show up in testing the entire application for current use case
  - After fixing a bug, build minimal test case exposing the bug and add to a library of regression tests in order to keep it from reappearing

Importance of Tests and Validation

- With a larger user base comes responsibility
  -> a test suite confirms available functionality
- No new code should break existing functionality
- Changes may have unintended side effects
- The more flexible a software is, the more potential for users to use it in unexpected ways
- Applications can fail on platforms due to broken compilers or system libraries
- Writing tests helps understanding a feature

Embedded Documentation

- Three types of documentation needed
  - Information for developers who want to add code
    -> Documentation of the API (e.g. via doxygen)
    -> Comments in the code that explain choices
  - Information for users that want to use a feature
    -> Reference manual for visible commands (can be automated and cross-linked with developer manual
  - Information for users that want to learn using a tool
    -> write tutorials and HOWTO segments
    -> often better written as standalone documents
- It can be helpful to write documentation first

Source Code Management

- Not only a way to archive sources, but a tool for communication between developers
- Distributed source code management makes concurrent development easier
- Work with feature branches and merge often
- Commit changes in small increments and do not combine unrelated changes in on commit
- Have consistent, documented “whitespace rules” and best enforce them before committing

The Bottom Line

- Many of these concepts and methods can help improve scientific software development
- **Important**: it is not the tools by themselves, but how they are used that makes the difference
- Fight the urge to take shortcuts and see the restrictions that modular and object oriented programming imposes as opportunities
- Finding the right balance is key to success
- Never underestimate the longevity of your code

Software Development Basics

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu