Advanced Software Development in Science

Dr. Axel Kohlmeyer

Research Professor, Dept. of Mathematics
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

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

a.kohlmeyer@temple.edu

Traditional Software Development "Waterfall Method"

- Fixed development phases
  - *Planning*: define deliverables, assesse risks, determine milestones and deadlines
  - *Implementation*: developers work on deliverables, development teams divide tasks among developers
  - *Integration*: deliverables are merged into alpha test version, conflicts in implementation are resolved
  - *Testing*: various stages of (internal) testing against documented requirements => release candidate
  - *Acceptance/Release*: repeat steps until accepted

Properties of the “Waterfall Method”

- Rigid: difficult to handle changing requirements
  => planning mistakes or updated requirements
  will cause significant delays
- Usable product only at the end of the cycle
- Encourages specialized teams for each phase
  => little exchange of knowledge and experience
- Favors fast implementation over code quality
- Requires well defined goals and deliverables
- Most effective for small teams and projects

Agile Software Development

- Increasing size and complexity of software projects expose limits of “waterfall” method
- Development of various techniques generally described with “Agile Software Development”:
  - Continuous Integration: code is always usable
  - Test driven development: testing becomes integral part of development; development becomes bugfix
  - Development sprints: small, incremental changes
  - Code review, pair programming: focus on code quality, changes are communicated early

Agile Development Goals

- More flexible development cycles
- Focus in software quality
- Modularity, code reuse, maintainability
- Constant, sustainable development progress
- Cross-function competent developers
- Maximize benefits from development tools
  - distributed source code management
  - automated unit, regression, & integration testing

Scientific Software Development Idiosyncrasies

- Scientific software is often developed to solve <u>specific</u> problems, not to generate revenue => less pressure to <u>prove</u> a feature is working
- The developer is often also the customer => superficial testing is considered adequate, since you have confidence in your capabilities => How can the software be wrong if it gives the right answer, anyway?
- Many developers have no formal training in software engineering, so they don't even know

Some More Scientific Software Development Idiosyncrasies

- There is little credit to be had for software development compared to <u>using</u> the software => any additional effort invested besides the minimum is reducing the competitiveness
- It is difficult to obtain funding directly for (non-commercial) scientific software development
- The bulk of the software development work is done by inexperienced people (students, post docs); advisers are not trained in management (of software development projects)

Even More Scientific Software Development Idiosyncrasies

- The correctness of a specific result is often not affected by code quality or efficiency
- A specific application may only be needed once
- Goals and tasks of scientific software for research are rarely well defined deliverables; they may change with how the science evolves
- Applications are often a complex composite of many units and it is thus difficult to test anything but the complete application

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

Linear Software Development

- One change is made after the other is complete
- Only one person can make changes at a time
- No need for source code management software:
  - Make a copy of the code
  - Add new features, test if working
  - Modified copy becomes new master version
- Sorce code management software can help through managing access and providing a per file change history (Example: RCS)

Concurrent Software Development

- Multiple developers work on copies checked out from a common managed repository
- Repository represents the canonical version; concurrent development, but is serialized when committing changes back into the repository
- Only commited changes generate a history
- Developer must merge changes from repository since checkout into local copy or they are lost
- Example: CVS (originally some scripts for RCS)

Non-linear Software Development

- Changes are worked on concurrently
  => need to use branches or multiple checkouts
- Branches generate independent commit histories; useful if long lived branch as the commit history provides insight into motivation
- Branches allow fine grained access control
- Merging can be complicated; source code management software can assist; especially when merging from a branch multiple times
- Examples: CVS and Subversion (SVN)

Distributed Software Development

- Distributed source code management software does not require access to canonical repository
  => multiple repositories including a local one
  => communicating changes means transferring data between repositories and merging
- Distinguish between local and remote branches
  => a local branch may “track” a remote branch
  => a local branch may be “pushed” to a remote
- What becomes the canonical version becomes a matter of agreement between developers

Distributed Source Code Management Software

- Popular: Git, Mercurial, Bazaar
- Can implement different development schemes
- Make branching and merging easy and fast
- Work on multiple branches from single working directory; switching between branches is easy
- Encourages frequent commits, commits in small logical units, work on short lived branches
- Downside: complexity
- Important: SCM is a means for communication

Advanced Software Development in Science

Dr. Axel Kohlmeyer

Research Professor, Dept. of Mathematics
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

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

a.kohlmeyer@temple.edu

MHPC
Master in High Performance Computing