SAND2010-7789C

The State of Trilinos Software Engineering

Recent Progress, Current Status, and Future Issues

Roscoe A. Bartlett

http://www.cs.sandia.gov/~rabartl/

Department of Optimization & Uncertainty Estimation

Trilinos Software Engineering Technologies and Integration Lead

Sandia National Laboratories

Trilinos Users Group Meeting 2010, Nov. 4, 2010

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Sandia National Laboratories

Part I

The Software Engineering Challenge in Computational Science & Engineering

and

The Role that Trilinos Can Play

Sandia National Laboratories

Vision for a Confederation of CSE Software?

Sandia National Laboratories

Factors Driving Increased Complexity in CSE Software

Progress in Computational Science & Engineering (CSE) is occurring primarily through the creation of greater varieties of increasingly more complex algorithms and methods

- **Discretization**: a) geometry, b) meshing, b) approximation, c) adaptive refinement, …
- **Parallelization**: a) parallel support, b) load balancing, …
- **General numerics**: a) automatic differentiation, …
- **Solvers**: a) linear-algebra, b) linear solvers, c) preconditioners, d) nonlinear solvers, e) time integration, …
- **Analysis capabilities**: a) embedded error-estimation, b) embedded sensitivities, c) stability analysis and bifurcation, d) embedded optimization, d) embedded UQ, …
- **Input/Output** …
- **Visualization** …
- ...

Complexity is also increasing due to new computer architectures (exascale):

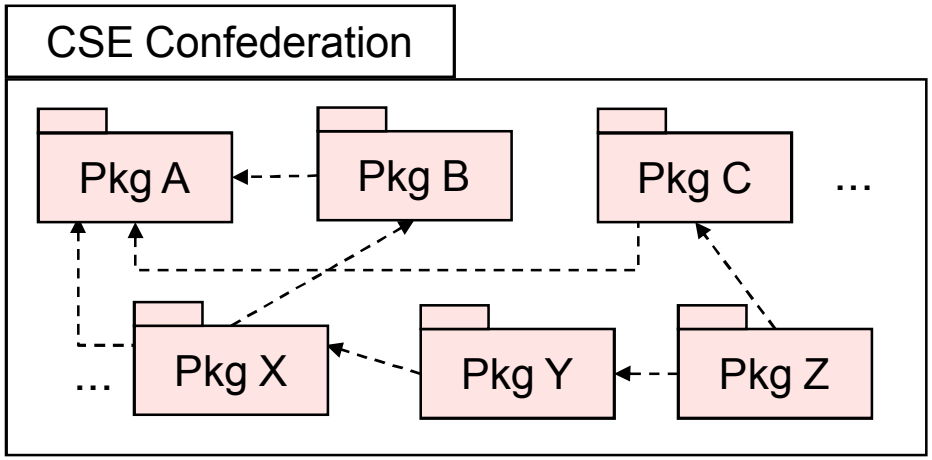
- Multi-core CPUs and GPUs => More complex programming and software architecture
- Decreased "mean time to failure" => More complex "robust" algorithms

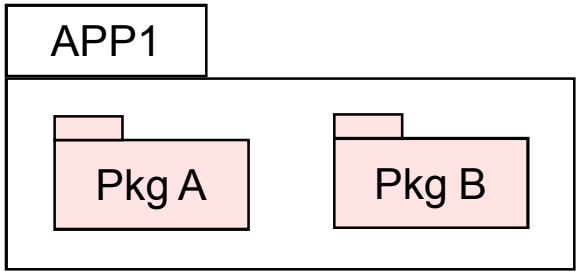
- Each technology requires specialized PhD-level expertise to implement
- Almost all technologies can be exploited to solve the full problem (i.e. design, V&V, QMU, …)
- Set of algorithms/software is too large for any single organization to create

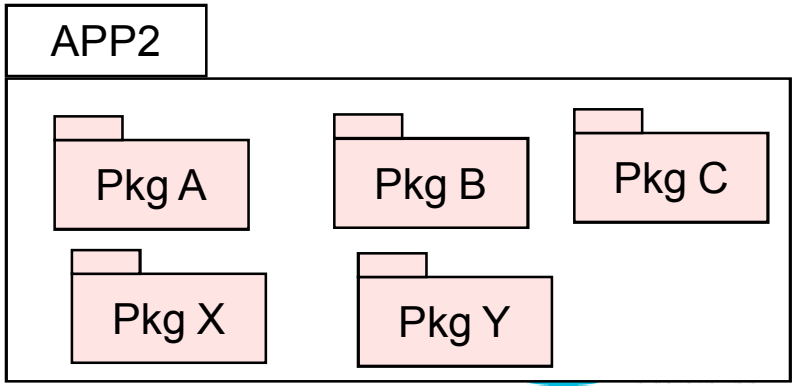
These treads will lead to greater than an order-of-magnitude increase in CSE software complexity which will require an order-of-magnitude increase in knowledge/skills/dedication/discipline in software design/integration/engineering!

The CSE Software Engineering Challenge

- Develop a confederation of trusted, high-quality, reusable, compatible, software packages/components including capabilities for:
  - **Discretization**: a) geometry, b) meshing, b) approximation, c) adaptive refinement, …
  - **Parallelization**: a) parallel support, b) load balancing, …
  - **General numerics**: a) automatic differentiation, …
  - **Solvers**: a) linear-algebra, b) linear solvers, c) preconditioners, d) nonlinear solvers, e) time integration, …
  - **Analysis capabilities**: a) embedded error-estimation, b) embedded sensitivities, c) stability analysis and bifurcation, d) embedded optimization, d) embedded UQ, …
  - **Input/Output** …
  - **Visualization** …
  - ...







Trilinos itself is a smaller example of this!

Obstacles for the Reuse and Assimilation of CSE Software

Many CSE organizations and individuals are adverse to using externally developed CSE software!

Using externally developed software can be as risk!

- External software can be hard to learn
- External software may not do what you need
- Upgrades of external software can be risky:
  - Breaks in backward compatibility?
  - Regressions in capability?
- External software may not be well supported
- External software may not be support over long term (e.g. KAI C++, CCA)

What can reduce the risk of depending on external software?

- Apply strong software engineering processes and practices (high quality, low defects, frequent releases, regulated backward compatibility, …)
- Ideally … Provide long term commitment and support (i.e. 10-30 years)
- Minimally … Develop Self-Sustaining Software (open source, clear intent, clean design, extremely well tested, minimal dependencies, sufficient documentation, …)

Some Lean Software Engineering Principles

Sandia National Laboratories

Overview of Lean Methods (7 Principles of Lean)

- Motivation for Lean Software Methods:
  - Lean Product Development (e.g. Toyota Product Development System)
- Seven Principles of Lean:
  1. Eliminate Waste
  2. Build Quality In
  3. Create Knowledge
  4. Defer Commitment
  5. Deliver Fast
  6. Respect People
  7. Optimize the Whole

Poppendieck, Mary and Tom. *Implementing Lean Software Development. Addison* Wesley, 2007

Lean Principle #1: Eliminate Waste

- Key to reducing complexity
  - Write less software!
  - Implement 20% of the code that satisfies 80% of requirement!
- The 7 wastes of software development:
  1. Partially done work (tests, documentation, release notes, deployed, etc. => “Done Done”)
  2. Extra features
  3. Relearning
  4. Handoffs
  5. Task switching
  6. Delays
  7. Defects!!!!!!!!! => 50% of effort on most projects spend fixing defects!

Lean Principle #2: Build Quality In

- Synchronize:
  - Write tests (unit, acceptance, verification, etc.) first => TDD
  - Integrate continuously => Used nested synchronization for larger sets
  - Find a defect? => “Stop the Line”
- Automate:
  - Mistake-proof through automation (e.g. Trilinos checkin-test.py tool)
- Refactor:
  - Keep code base clean and simple => Agile (Emergent) Design:
    - no duplication
    - clean design
    - minimize complexity

Lean Principle #5: Deliver Fast

- Work in small batches:
  - Reduce project size (e.g. smaller Stories)
  - Shorten release cycles
  - Stabilize, repeat
- Limit work to capacity:
  - Use pull scheduling
  - Limit queue sizes
  - Accept no work unless there is room
- Focus on cycle time, not utilization:
  - Full (and over) utilization reads to thrashing and waste!

Lean Principle #6: Respect People

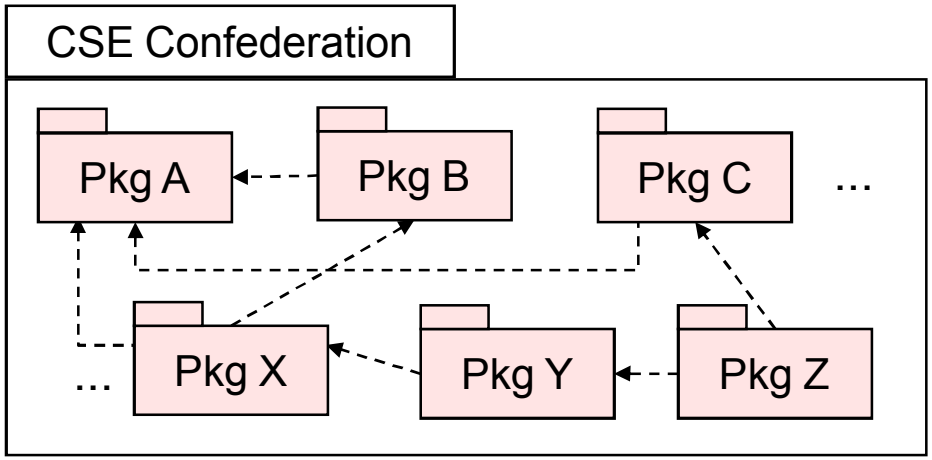
- Train team leaders/supervisors
  - Provide the necessary training and then trust
- Move responsibility and decision making to the lowest possible level:
  - Let teams design/define their own processes
  - Provide the resources & tools to succeed
- Foster pride in workmanship:
  - No sloppy workspaces
  - No sloppy work practices
  - No impossible deadlines
  - Give proper time for design, testing and refactoring
  - Automate routine tasks
  - Never ask for sloppy work to meet a deadline!

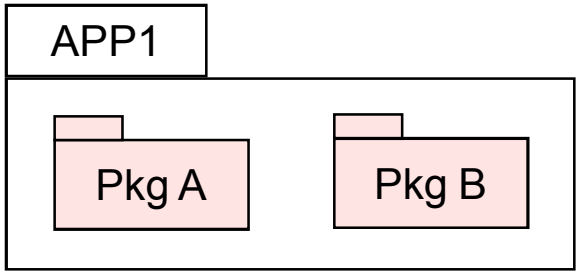
Realizing a Confederation of CSE Software

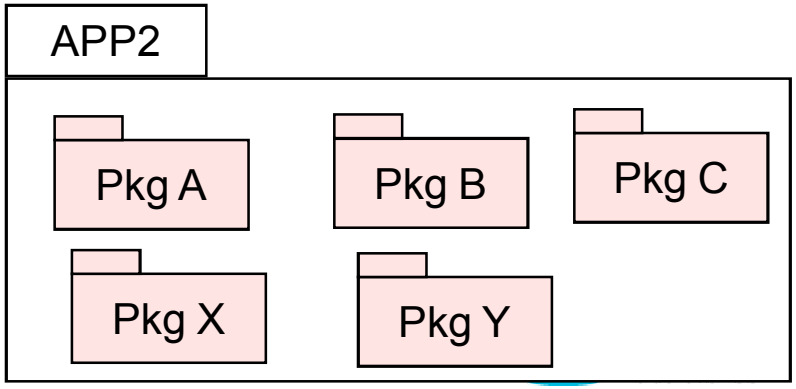
Sandia National Laboratories

The CSE Software Engineering Challenge

- Develop a confederation of trusted, high-quality, reusable, compatible, software packages/components including capabilities for:
  - **Discretization**: a) geometry, b) meshing, b) approximation, c) adaptive refinement, …
  - **Parallelization**: a) parallel support, b) load balancing, …
  - **General numerics**: a) automatic differentiation, …
  - **Solvers**: a) linear-algebra, b) linear solvers, c) preconditioners, d) nonlinear solvers, e) time integration, …
  - **Analysis capabilities**: a) embedded error-estimation, b) embedded sensitivities, c) stability analysis and bifurcation, d) embedded optimization, d) embedded UQ, …
  - **Input/Output** …
  - **Visualization** …
  - …







Trilinos itself is a smaller example of this!

Requirements/Challenges for a Confederation of CSE Codes

- Software quality and usability
  => Design, testing, training, standards, collaborative development
- Building the software in a consistent way and linking
  => Common build approach (e.g. CMake)
- Reusability and interoperability of software components
  => Incremental Agile design, runtime resource management, …
- Critical new functionality development
  => Closer development and integration models
- Upgrading compatible versions of software
  => Frequent fixed-time releases, regulated backward compatibility
- Safe upgrades of software
  => Regulated backward compatibility, reducing defects
- Documentation, tutorials, user comprehension
  => SE education, better documentation and examples
- Self-sustaining software (open source, clean design, clean implementation, well tested with unit tests and system verification tests, minimal dependencies)
  => Anyone can maintain it!
- Long term maintenance and support
  = > Stable organizations, stable projects, stable staff

Sandia National Laboratories

PART II

Recent Progress and Current status of Trilinos SE

Sandia National Laboratories

Recent Progress in Trilinos SE Infrastructure

- Moved to Git version control system
- External repositories and add-on Trilinos packages
  - Partitioning of copyrighted and non-copyrighted packages
  - Scalable no-direct growth (LIMEExt, TerminalPackages)
- Support for deprecated warnings for GCC with macros
- Improvement in release processes:
  - More frequent releases: Has lead to less instability of releases
  - See Jim's talk
- Testing improvements (see later slide):
  - Introduction of CATEGORIES keyword (e.g. BASIC, NIGHTLY, etc.)
  - Better pre-push and post-push CI Testing, faster computers
- More extensive/safer Teuchos memory management classes:
  - www.cs.sandia.gov/~rabartl/TeuchosMemoryManagementSAND.pdf
- SIERRA Trilinos Almost Continuous Integration process:
  - Nightly testing (< 48 hour delay) of a lot of Trilinos (Teuchos through MOOCHO) on many platforms (GCC, Intel, AIX, Pathscale, PGI, etc.)
  - SIERRA takes snapshots of Trilinos for releases
- Greater Trilinos development stability:
  - Allow for daily integration testing and daily updating of customer APPs

PART III

Immediate Needs in Trilinos Software Engineering

Sandia National Laboratories

Immediate needs and Related Efforts in Trilinos SE

- Generalize and externalize the Trilinos CMake/CTest/CDash system
  - Allow other projects to fully exploit the Trilinos SE infrastructure
  - Will be used by projects like NEAMS, CASL and perhaps others
- Centralize information and keep up to date on Trilinos websites
  - Do Google site searches first to answer questions.
- Further improvements in Trilinos release-related efforts and processes
  - Automated tarball testing
  - Automated installation testing
  - Move (almost) all release-related work **before** the branch
  - See Jim's talk …
- More effort/discipline in maintaining Trilinos testing processes:
  - Example: TrilinosDriver failures on s909348
  - Example: Failures uploading and timing out of coverage tests
- Improvements in Testing (see slide)
- Sub-Package Support (see slide)
- Need more Trilinos Framework Staff!

Testing Improvement Needs

Sandia National Laboratories

A Truism Related to Software Features and Testing

Murphy's Law for Software:

> "For any attribute you claim your software has, if you don't have strong automated tests to provide strong evidence for that property, you can almost guarantee that the property will not exist just when it will do the most damage."

Example:

Customer: "Your algorithm X does not scale very well"

Developer: "Nonsense, I tested the scalability myself just a few months ago"

Customer: "Did you test scalability on the exact release version I am using?"

Developer: "No, I did not have time to do the testing again."

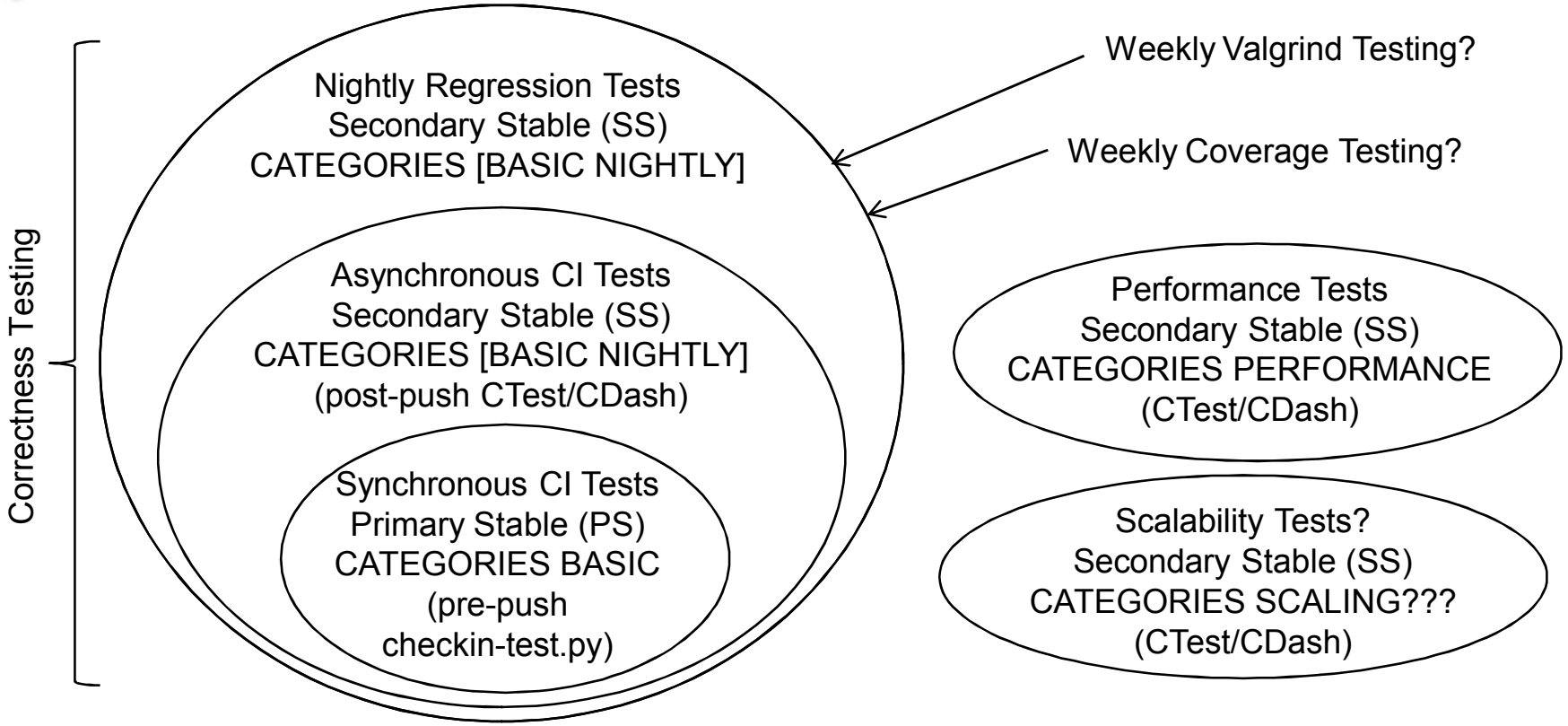
Customer: "What could have changed?"

Developer: "Some code somewhere obviously or perhaps something else?"

Example of claims about software:

- "The software is efficient"
- "The software scales"
- "The software is well tested"
- "The software works"

Testing Improvements Needed



Status of each testing category:

- ❑ Synchronous CI Tests: Better use of checkin-test.py script; need faster better (unit) tests
- ❑ Asynchronous CI Tests: Automatic, fast feedback (e.g. < 20 min); need better notifications
- ❑ Nightly Regression Tests: Linux/Mac, GCC/Intel; need better coverage and more platforms
- ❑ Performance Testing: Just a few Teuchos tests (see TSDM10); need
- ❑ Weekly Coverage Tests: Not running on all of Trilinos; has recurring failures
- ❑ Weekly Valgrind Tests: Not currently running; need a dedicated machine to run weekly
- ❑ Scalability Tests: Not even defined, not even CATEGORIES support yet

Trilinos CMake Sub-Package Support

Sandia National Laboratories

CMake Sub-Package Architecture: Motivation

Existing package dependency logic can enable many more packages than is needed for pre-checkin testing

Example: Enable Tpetra

```
$ checkin-test.py --enable-packages=Tpetra –configure
```

- Enabled packages (libraries) (28/52): Teuchos, RTOp, Kokkos, Epetra, Zoltan, Shards, Triutils, Tpetra, EpetraExt, Thyra, Isorropia, AztecOO, Galeri, Amesos, Pamgen, Ifpack, ML, Belos, Stratimikos, Meros, Anasazi, RBGen, Sacado, Intrepid, NOX, Rythmos, MOOCHO, Sundance
- Enabled packages (tests/examples) (10/52): Tpetra, Belos, Stratimikos, Anasazi, RBGen, NOX, Rythmos, MOOCHO, Sundance

=> Problem: Stratimikos, Rythmos, MOOCHO, and Sundance don't execute one line of Tpetra code!

- General Problem: Current CMake build system does not respect the existing package partitioning

Software Engineering Theory about Packaging

Package Cohesion OO Principles:

- REP (Release-Reuse Equivalency Principle): The granule of reuse is the granule of release.
- CCP (Common Closure Principle): The classes in a package should be closed together against the same kinds of changes. A change that affects a closed package affects all the classes in that package and no other packages.
- CRP (Common Reuse Principle): The classes in a package are used together. If you reuse one of the classes in a package, you reuse them all.

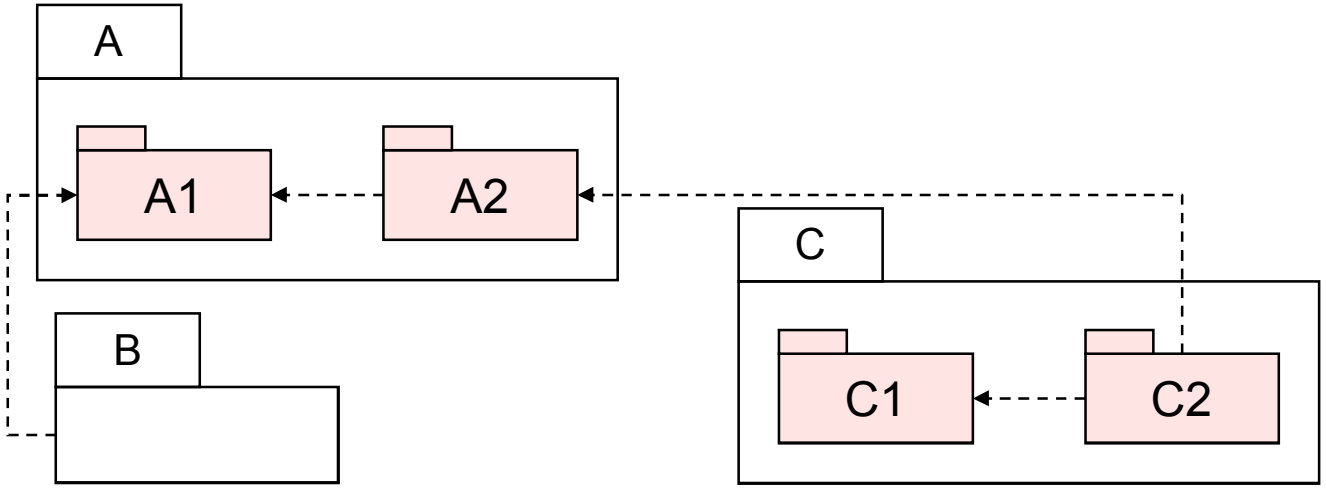
Package Coupling OO Principles:

- ADP (Acyclic Dependencies Principle): Allow no cycles in the package dependency graph.
- SDP (Stable Dependencies Principle): Depend in the direction of stability.
- SAP (Stable Abstractions Principle): A package should be as abstract as it is stable.

Problem: Many Trilinos packages violate the SE packaging principles most importantly the CRP

Source: Martin, Robert C. *Agile Software Development (Principles, Patterns, and Practices)*. Prentice Hall, 2003

CMake Sub-Package Architecture: The Idea



- Trilinos packages: More natural feature/social/user/documentation collections
- Trilinos sub-packages: Dependency-management SE packages (hidden from user)
- Speeds up pre-push rebuilds and testing with checkin-test.py tool
- Provided greater control over feature selection
- Helps to minimize superficial entangling dependencies
- Minimizes the number of top-level packages
- Hides complexity form the user
- However, some SE packages will still be needed due to dependency issues
- Git allows us to move files around into different directories!
- See Trilinos Framework Backlog Item 4644!

PART IV

Longer Term Issues in Trilinos SE

Sandia National Laboratories

Updated Trilinos Life-Cycle Model?

Goals for an updated Trilinos Life-cycle Model

- Allow pure research and applied research with a realistic path to productionization
- Provide smooth low-effort transitions from research to production phases
- Provide maximum confidence with low cost (for research results and then real users)
- Allow the use of Lean/Agile practices and processes along the way

Proposed Phases for new Trilinos Life-cycle Model

1) Purely Experimental Code

2) Research Stable Code

3) Production Growth Stable Code

4) Production Maintenance Stable Code

See Trilinos Framework Backlog Item 4837

Proposed Trilinos Life-cycle Model: Research Phases

1) Purely Experimental Code:

- **Not** developed in a Lean/Agile consistent way
- Could actually be declared to be Secondary Stable code with respect to CI and nightly Trilinos testing but in general would be considered to be Experimental code in Trilinos
- Does not provide sufficient unit (or otherwise) testing to prove correctness
- No one should use it for anything important (not even for research results but in the current CS&E environment publication would be allowed)
- Should *not* go out in general releases of Trilinos
- Does *not* provide a direct foundation for creating production-quality code

2) Research Stable Code:

- Developed in a Lean/Agile consistent way
- Strong unit and verification testing (i.e. proof of correctness) written while the code/algorithms are being developed
- Could be Primary Stable or Secondary Stable code in Trilinos
- Does **not** generally have good examples, documentation, etc.
- Appropriate to be used by "expert" users
- Appropriate to be part of a general release
- Appropriate to be used in customer codes
- Would tend to provide for regulated backward compatibility but not in all cases
- Can provide the foundation for creating production-quality code

Proposed Trilinos Life-cycle Model: Production Phases

3) Production Growth Stable Code:

- Includes all the good qualities of "Research Stable Code" plus ...
- Improving validation of user input errors and better error reporting
- Improving formal documentation (Doxygen, technical reports, etc.)
- Improving examples, tutorial material, etc.
- Optional refactoring of the code structure and user interfaces to make more consistent, easier to maintain (should be needed for Agile designed software)
- Maintains rigorous regulated backward compatibility with few (if any) truly incompatible changes with new releases
- Expand usage in customer codes
- Appropriate to turn over to a maintenance support team at any time

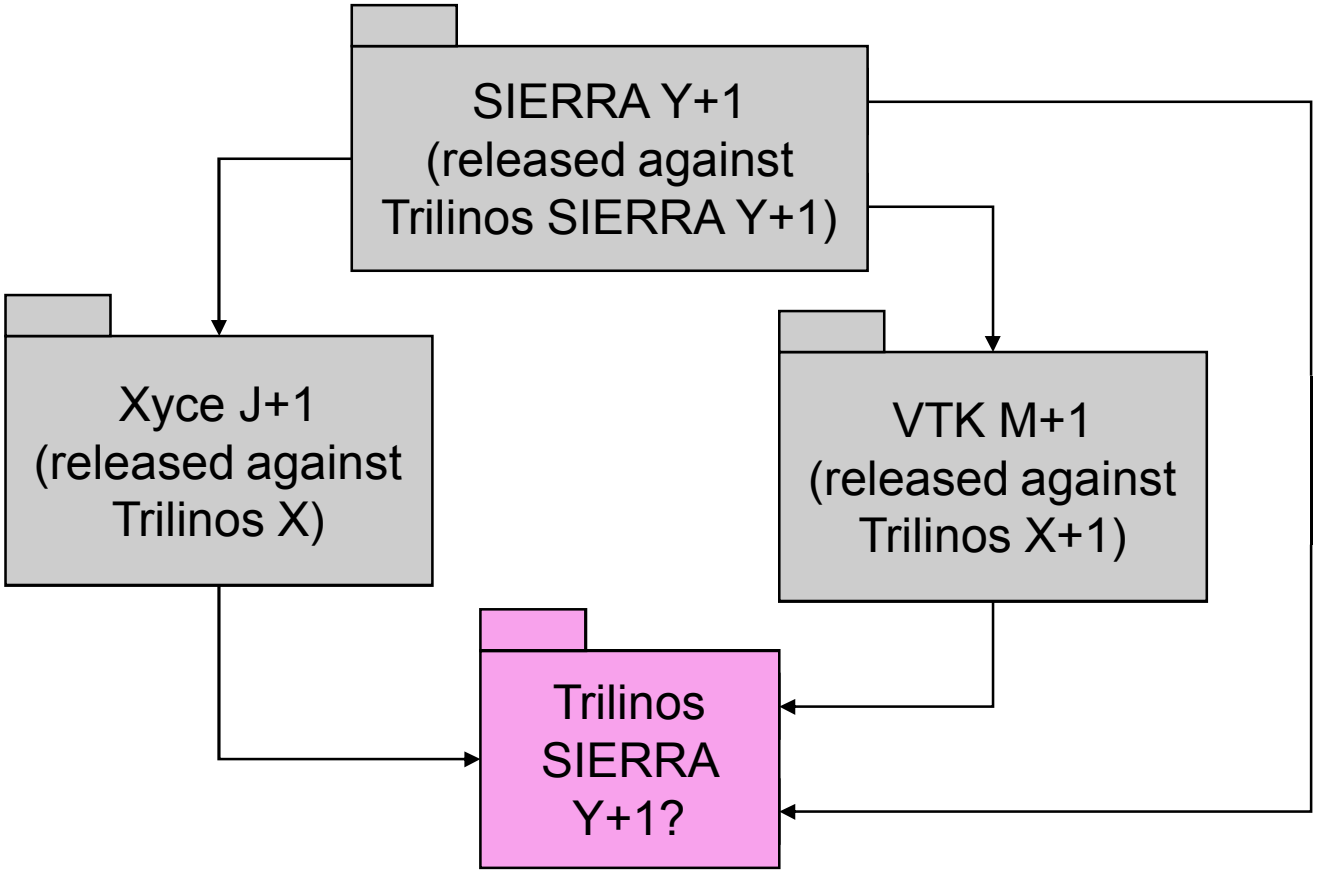
4) Production Maintenance Stable Code:

- Includes all the good qualities of "Production Growth Stable Code" plus ...
- Primary development only includes bug fixes and performance tweaks
- Maintains rigorous backward compatibility with typically no deprecated features
- Could be maintained by parts of the user community if necessary

Regulated Backward Compatibility

Sandia National Laboratories

Example of the Need for Backward Compatibility



Multiple releases of Trilinos presents a possible problem with complex applications

Solution:

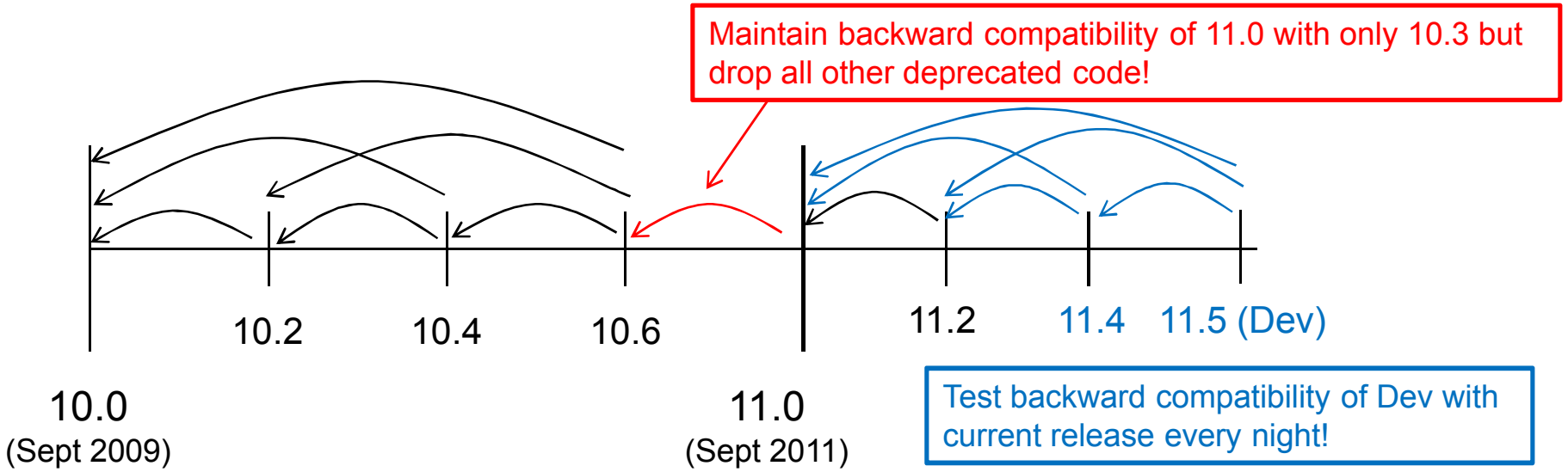
=> Provide sufficient backward compatibility of Trilinos X through Trilinos SIERRA Y+1

Backward Compatibility Considerations

- Backward compatibility is critical for:
  - Safe upgrades of new releases
  - Composability and compatibility of different software collections
- Maintaining backward compatibility for all time has downsides:
  - Testing/proving backward compatibility is expensive and costly
  - Encourages not changing (refactoring) existing interfaces etc.
    - => Leads to software "entropy" which kills a software product
- A compromise: Regulated backward compatibility (Trilinos approach)
  - Maintain a window of "sufficient" backward compatibility over major version numbers (e.g. 1-2 years)
  - Provide "Deprecated" compiler warnings
    - Example: GCC's __deprecated__ attribute enabled with –DTrilinos_SHOW_DEPRCATED_WARNINGS:BOOL=ON
  - Drop backward compatibility between major version numbers
  - [Future] Provide strong automated testing of Trilinos backward compatibility

Regulated Backward Compatibility in Trilinos

- Trilinos Version Numbering X.Y.Z:
  - X: Defines backward compatibility set of releases
  - Y: Major release (off the master branch) number in backward compatible set
  - Z: Minor releases off the release branch X.Y
  - Y and Z: Even numbers = release, odd numbers = dev
    - Makes logic with Trilinos_version.h easier
- Backward comparability between releases
  - Example: Trilinos10.6 is backward compatible with 10.0 through 10.4
  - Example: Trilinos 11.X is not compatible with Trilinos 10.Y



Example: Major Trilinos versions change every 2 years with 2 releases per year

Collaborative Development (e.g. Code Reviews)

and

Static Analysis Tools

Sandia National Laboratories

Collaborative Development (e.g. Code Reviews)

- Importance of Collaborative Development Practices ("Code Complete 2nd")
  - Testing alone will only achieve the detection of 60% or less of defects (Jones 2000)
  - Formal code reviews alone achieve 60% defect detection rates
  - Per defect, reviews are 20 times cheaper than black-box testing (Freedman and Weinberg 1990).
  - High-volume beta testing (> 1000 sites) achieves less than 85% defect detection.
  - Formal code reviews combined with testing can achieve 95% or higher defect detection/removal (Jones 200)
  - You can't afford not to implement some type of code review process!
- Approaches to doing effective collaborative development ("Code Complete 2nd")
  - Formal code reviews
  - Inform reviews (e.g. code reading, walk-throughs, dog-and-pony shows, …)
  - Pair programming
- Issues that are well addressed by code reviews:
  - Readability, comprehensibility, change tolerance, duplicate code, design quality (good naming, good use of OO patterns and principles, etc.)
- Issues not well addressed by code reviews ("Implementing Lean Software Development"):
  - Enforcing low-level coding standards => Use static analysis tools instead
  - Catching low-level defects => Use TDD and unit testing instead

Integrated Static Analysis Tools in Development Environment

- Leaning and improving quality through automated tools and continuous feedback?
  - Example: Writing better C++ using g++ -Wall –pedantic –ansi …
  - We don't expect everyone to be experts in C++ to write C++ code because modern C++ compilers provide lots of feedback and help in learning C++.
- Tools not integrated into development environment are ignored!
  - Example: What is the coverage of your Trilinos package today?
- Use integrated tools to help flag and enforce coding standards, etc.:
  - Example of standards to be checked for automatically:
    - "C++ Coding Standard": Sutter and Alexandrescu
    - "Thyra Coding and Documentation Guidelines"
      - http://www.cs.sandia.gov/~rabartl/ThyraCodingGuideLines.pdf
  - Goals of static analysis tools: Teach, enforce standards, consistency, etc.
- Examples of tools that can be integrated into development environment:
  - Separate source analysis tools: AStyle, Google cpplint.py
  - Integrated with GCC compiler:
    - Mozilla Dehydra: https://developer.mozilla.org/en/Dehydra
      - GCC 4.5.x+ plugin that executes user-defined checks as part of compilation!

Official Trilinos Developers Toolset

Sandia National Laboratories

Official Trilinos Developers Toolset: Idea and Motivation

- Idea: Define a suite of standard build and other tools along with simple global install script
- Candidate list of software:
  - GCC 4.X.Y (Fortran or no Fortran?)
  - Gold ??? (fast linking)
  - Open MPI ???
  - CMake 2.8.X
  - Git ???, eg ???
  - CLAPACK ???
  - Boost ???
  - Doxygen ???
  - Dot ???
- Motivation:
  - Reduce variability in development and testing for different developers
    - Turn on strong warnings and warnings as errors!
  - Simplify setup of new Trilinos development machines
  - Allow more code to be elevated to Primary Stable Code (e.g. boost)

Official Trilinos Developers Toolset: Install scripts

Provide global install script:

$ Install-trilinos-toolset.py –do-all –install-dir=/home/trilinos/install

- Checks out tarballs from Trilinos3PL CVS repository
- Installs all software in single bin, lib, and include directories
- Uses separate install scripts like install-cmake,py, install-git.py etc.
- Would only support basic Linux (perhaps Unix) and Mac computers (not Windows)

ToDo:

- Decide what software should be included
- Decide on versions of all the software packages
- Refactor existing install-git.py and install-cmake.py to enable faster development of simple install steps
- Get software and write basic install scripts and global install script
- Beta users to work out bugs
- Deploy across all Trilinos developers
- Turn on warnings as errors!
- Enjoy more a stable development environment!

Partitioning of Trilinos Git Repository for Sustained Growth?

Sandia National Laboratories

Miscellaneous Areas of Needed Improvement and Progress

- Code coverage (see TrilinosCMakeQuickstart.txt)
  - $ ./do-configure -DTrilinos_ENABLE_COVERAGE_TESTING:BOOL=ON
  - $ make dashboard
- Memory checking (see TrilinosCMakeQuickstart.txt)
  - $ env CTEST_DO_MEMORY_TESTING=TRUE make dashboard
    - Need a trimmer test suite to allow valgrind to run
- Namespace safety
  - Don't pollute the global namespace, no 'using namespace ANTHYING'
- Strong warnings and warnings as errors
  - Need a standard version of GCC and MPI first (Official Trilinos Toolset)
- Doxygen documentation (Need automated testing of some type)
- Improving exception safety (basic guarantee, strong guarantee, and no-fail guarantee and memory leaks)

THE END

Sandia National Laboratories