Maintaining the Stability of Trilinos Dev

Stable vs Experimental Code

Roscoe A. Bartlett
Department of Optimization & Uncertainty Estimation

Sandia National Laboratories

Trilinos User Group Meeting, October 23, 2008
Motivations for Improving Stability of Trilinos Dev

• Support deep stacks of vertically integrated Trilinos packages with production APPs
  – Algorithm Integration Project
  – Many others ...

• Support tighter coupling and co-development with production APPs
  – SIERRA toolkit packages (STK_Mesh, STK_IO, ...)
  – Replace SIERRA framework code with Trilinos code (Teuchos::ParameterList, ...)

• Support more frequent, safer, higher quality, lower risk releases of Trilinos

• Improve overall development productivity and software quality

See:
  Trilinos/doc/DevGuide/TrilinosSoftwareEngineeringImprovements.doc
Challenges of APP + Trilinos Release and Dev Integration

**Problems**
- APP developers sometimes break APP + Trilinos Dev New
- APP + Trilinos Dev inherits instability of APP and Trilinos development lines

**Improvements**
- Make Trilinos Dev backward compatible with Trilinos Release
  => Minimize need to refactor and ifdef
- Improve stability of Trilinos Dev
- Improve stability of APP VOTD
SIERRA + Trilinos Integration: Opportunities and Challenges

• SIERRA Framework Developers would like to consider tighter integration with Trilinos:
  – Move new SIERRA toolkits packages into Trilinos
    • STK_Mesh
    • STK_IO?
    => Make these available for rapid production and other projects
  – Develop the FEI through Trilinos instead of a SIERRA TPL
    => Allow FEI to be updated more frequently
  – Replace SIERRA code with Trilinos code:
    • Teuchos::ParameterList
    • Intrepid
    • Phalanx
    => Reduce duplication and increase Trilinos impact

• Challenge: Tighter integration of APP and Trilinos does not fit well into current APP + Trilinos Release and Dev model!
APP developed only against Trilinos Dev

Trilinos Head

APP Head

Future of SIERRA + Trilinos Integration?

Testing: APP VOTD + Trilinos Dev

Supported with continuous integration testing!

• All changes are tested in small batches
• Low probability of experiencing a regression
• Less computing resources for testing
• Regressions and flagged immediately by APP developers
• Can support tighter integration efforts
• Supports rapid development of new capability from top to bottom
• Requires Trilinos to be more stable
• Other issues arise as well
Lean/Agile Software Engineering Principles

- High quality software is developed in small increments and with sufficient testing in between sets of changes.
- High quality defect-free software is most effectively developed by not putting defects into the software in the first place (i.e. code reviews, pair programming etc.).
- High quality software is developed in short fixed-time iterations.
- Software should be delivered to real (or as real as we can make them) customers is short intervals.
- Ruthlessly remove duplication in all areas.
- Avoid points of synchronization. Allow people to work as independently as possible and have the system set up to automatically support this.
- Most mistakes that people make are due to a faulty process/system (W. Edwards Deming).
- Automation is needed to avoid mistakes and improve software quality.

Lean/Agile Methods: Development Stability

Common Approach
NOT AGILE!

Problems
• Cost of fixing defects increases the longer they exist in the code
• Difficult to sustain development productivity
• Broken code begets broken code (i.e. broken window phenomenon)
• Long time between branch and release
  – Difficult to merge changes back into main development branch
  – Temptation to add “features” to the release branch before a release
• High risk of creating a regression
Lean/Agile Methods: Development Stability

The Agile way!

Advantages
- Defects are kept out of the code in the first place
- Code is kept in a near releasable state at all times
- Shorten time needed to put out a release
- Allow for more frequent releases
- Reduce risk of creating regressions
- Decrease overall development cost
• Analogy: United States of America
  – Federal Government vs State roles and responsibilities

• Trilinos Analogy
  – Services provided by the framework (federal) for the packages (states)
  – Other services packages (states) provide for themselves
  => This as been described as a one-way street! Framework => Packages

• What about package responsibilities to the framework and other packages?

• Analogy: United States of America
  – Federal government imposes standards and requirements on States
    • Example: States can not deny voter rights in local elections
  – States have to support other states and the federal government
    • Example: States have to pay taxes to support the federal government
    • Example: If Florida is invaded by Cuba, all states will provide soldiers

• Trilinos package developers have extra responsibilities by being in Trilinos!
  => We need a two-way street! Framework ⇔ Packages
Trilinos “Stable” vs “Experimental” Code: Defined

• “Stable” Code and Tests:
  – “Stable” code meets one or more of the following criteria:
    • Represents an important capability being used by an existing customer in a release of Trilinos, or
    • Represents a new capability that the authors are willing to stand behind (as defined below) and is being targeted for the next release
  – “Stable” code/tests are expected to be kept working at all times. There should be little excuse for breaking “Stable” code on the primary development platform(s).
  – “Stable” code should be developed from the start and maintained to be highly portable.
  – “Stable” code should be maintained at the highest quality as defined by Lean/Agile software engineering principles.

• “Experimental” Code and Tests:
  – By definition, all remaining code that is not “Stable” code.
  – Represents fundamental research and may be developed with informal low-quality software practices.
  – Any code that has a direct and mandatory dependency on any “Experimental” code must also be considered to be “Experimental” code.
  – Developers should try to avoid depending on other “Experimental” code because it is likely to be unstable and break frequently.
  – “Experimental” code should be protected behind ifdefs with macOS that must be defined in order to be built.
Trilinos “Stable” vs “Experimental” Code: Goals

• Allow crazy and impulsive algorithms research with “Experimental” Code
  – Conducted within Trilinos
  – Benefit from ready to use “Stable” building blocks
  – Take advantage of everything the Trilinos environment has to offer

• Maintaining “Stable” core allows:
  – Other “Experimental” research efforts can remain highly productive because their foundation is not constantly breaking
  – New requirements from “Stable” code needed to drive “Experimental” research code development can be rapidly developed and integrated in real time

• Partitioning off “Experimental” code from “Stable” code
  – Avoid the problem of a top-heavy overly strict environment which does not allow for rapid research investigations.

• By keeping “Stable” code in a near releasable state, we allow for fast and frequent releases of Trilinos.

• Summary: We can have our cake and eat it too!
Sub-categorizations of “stable” code based on dependencies:

- “Primary Stable” code is “Stable” code that only depends on:
  - C, and C++ compilers
  - Fortran 77 compiler
  - BLAS and LAPACK
  - MPI

- “Secondary Stable” code is “Stable” code with additional dependencies such as:
  - SWIG/Python (i.e. PyTrilinos)
  - Fortran 2003+ (i.e. ForTrilinos)
  - External direct sparse solvers like UMFPACK, SuperLU, etc. (i.e. Amesos adapters)
  - ...

“Stable” code in one package can only depend on “Stable” code in other packages.

“Stable” code should by default only build “Primary Stable” code.

Enabling “Secondary Stable” code should require explicit configure-time options.
“Synchronous Continuous Integration”: Software is integrated and tested locally before it is checked in by performing the following in rapid succession:

- Do a VC update
- Rebuild all affected code
- Rerun the “precheckin” test suite
- If there are *any* failing tests
  - Fix the code, or
  - Investigate why the code fails, or
  - Do something else to make sure it is okay to check in.
  - Don’t just check in broken code and/or broken tests!
- If all the affected code and tests build and pass
  - Quickly check in the changes using one atomic checkin
Asynchronous Continuous Integration: Software is integrated and tested on a CI server after it is checked in by performing the following:

- Developers do basic/incomplete testing (i.e. without doing full “synchronous continuous integration”)
- Developers check in code
- Continuous integration (CI) server periodically runs:
  - Does VC update (on a fixed schedule or when changes are detected)
  - Does a full integration build and runs the integration test suite
- If the build or any tests fail
  - An email notification is sent out to some group of people alerting them

- Developers fix problems ASAP!
General Practices Related to “Stable” Code

- Stability and portability as the highest goals
- Maintained in a “done” (i.e. close-to-releasable) state
  - up-to-date tests, examples, documentation
- Compiled with high warning levels and treat warnings as errors
  - Portability of the software!
  - Many users want to compile their codes with this and Trilinos is a problem
- Backward compatibility (one major release or more) is a high priority
- Every Trilinos developer’s responsibility to help maintain stability and integrity
- Built every night on a variety of different platforms & compilers
  - High priority on fixing broken builds first and then on fixing broken tests
- 100% passing test policy for all “Stable” code on our primary development platforms!
  - Primary development platforms include:
    - Linux + gcc ??? (high warning levels and warnings as errors)
    - Mac OSX + gcc ??? (lesser warning levels?)
- Goal of 100% passing tests on other auxiliary platforms as well
  - Efforts to fix failing builds and tests on auxiliary platforms will take place in an auxiliary development loop that runs behind the efforts on the primary development platforms
  - Examples of auxiliary platforms: Intel compilers, PGI compilers, IBM compilers, Pathscale compilers, SUN compilers, ...
General Practices Related to “Experimental” Code

• Other Trilinos developers have little-to-no direct responsibility to maintain “Experimental” code
  – However, should still consider the impact their changes will have *before* they check in
  – Example: Use a script to automatically change the name in all source files

• Remove “Experimental” code segments and even entire files for releases
Promotion of “Experimental” Code to “Stable” Code

TBD
“Stable” vs “Experimental” Code: Continuous Integration

• “Primary Stable” code
  => Use “synchronous continuous integration” before every checkin
    • Steps can be skipped based on developer discretion
  => Provide driver tools to make this easy!

• “Secondary Stable” code
  => Use “synchronous continuous integration” for any “Primary Stable” code
  => Rely on “asynchronous continuous integration” for testing other “Secondary Stable” code
    • Respond to failed builds/tests ASAP!
  => Or, build and test on Central Build Server to test affected “Secondary Stable” code before checkins

• “Experimental” code
  => No pre-checkin procedure for “Experimental” code
  => Use “synchronous continuous integration” for any “Primary Stable” code
Synchronous Continuous Integration Checkin Procedure

A) Start filling out the checkin checklist message in a temporary text file
B) Do a VC update to get all current changes
C) Configure Trilinos to enable all “Primary Stable” code that depends on your code:
   – Without any “Secondary Stable” or “Experimental” code enabled
   – Test serial + debug (-pedantic), and mpi + optimized (high warning levels and warnings as errors)
D) Rebuild and rerun the “pre-checkin” test suite (high working levels and warnings as errors)
   – If there are *any* tests fail, fix the code, investigate why the tests are failing, etc ...
   – **DO NOT UNDER ANY CIRCUMSTANCE EVER CHECK IN CODE THAT DOES NOT BUILD!**
E) Finish filling out the checkin checklist message (while rebuilt/retest is running).
F) If the rebuild/restest passes and (i.e. all tests pass, 0 test fail), then:
   – Quickly do a ‘cvs –nq update –dP’ to see if there are any new changes
     • If you see changes that are worrisome, go back to step ‘B’ and repeat.
   – Otherwise, go ahead and check in
     • Do checkin in one global atomic checkin using the checkin with ‘cvs commit –F checkin_message’.
G) Otherwise, abort the checkin and then do either:
   – Backup your changes to keep them safe (e.g. using ‘tar –czvf Trilinos.date.tar.gz Trilinos’ or something and ‘scp’ the file to another machine).
   – Or, Try to resolve the problems and get the code to build and get all of the tests to pass.

NOTE #1: Any and all of the above steps can be bypassed by the developer
NOTE #2: A tool (Cmake-based) must be available to perform all of the above steps
NOTE #3: The fallback is to rely on “asynchronous continuous integration”
NOTE #4: Changes that break down-stream code are immediately caught *before* checkin
“Stable” vs “Experimental” Code: Daily Integration Testing

• Testing of “Primary Stable” Code
  – Tested on primary testing servers
  – Tested on other auxiliary platforms (as many as we can get)
  – Results reported to “Primary Stable” code dashboard section
  – Tested as part of APP + Trilinos Integration testing (i.e. Charon, SIERRA, Alegra, ...)

• Testing of “Secondary Stable” Code
  – All “Secondary Stable” code with all dependencies tested on a central testing server
  – Targeted subsets of “Secondary Stable” code tested on targeted auxiliary platforms
  – Tested as part of APP + Trilinos Integration testing (i.e. Charon, SIERRA, Alegra, ...)

• Nightly testing of “Experimental” code
  – Performed entirely on package teams computing resources
  – Takes advantage of easy-to-create testing drivers (i.e. Cmake/CTest)
  – Results posted to central dashboard separate from “Stable” code results
  – Tested as part of APP + Trilinos Integration research testing (i.e. Charon, SIERRA, Alegra, ...)


• All “Stable” code should have 100% passing tests 100% of the time on the primary development platforms as the norm instead of the exception.

• Achieving 100% passing tests on auxiliary development platforms is also a priority but is done in a secondary development loop.

• A failing test on any testing platform should be addressed and be made to pass or be disabled using the following algorithm:
  – Fix the test in the strongest way possible
  – Or, loosen the “strength” of test to get it pass on that specific platform (i.e. by loosing a platform-specific tolerance)
  – Or, disable the test and submit a new item to the sprint or product backlog (e.g. Bugzilla bug report) so that it can be prioritized and fixed later
  – Or, remove the test and all of the associated code related to it
Motivations for a 100% Passing Test Policy for “Stable” Code

Why is 100% passing tests important?

• Package Y (reference package):
  – “Broken Window” Phenomenon
    => One broken test begets others
  – Zero (0) is singularly different than 1 or X failing tests
    => People take notice of “all passed” vs “failed”
  – ‘M’ failing tests is not much different than ‘N’ failing tests
  – 100% passing tests clear measure of the code health
  – 100% passing test suite is unbiased criteria for code checkins
  – 100% passing test suite is an unbiased measure for if any code has been broken after a checkin
  – Code coverage less meaningful when there are failing tests

• Package X (up-stream package being used by Package Y)
  – 100% passing test suite for Package Z provides a clear means to determine if changes in Package X break anything.

• Package Z (down-stream package that uses Package Y)
  – 100% passing test suite for Package Y gives Package Z developers confidence that they can depend on and trust the code in Package Y.

• Bottom Line:
  – 100% passing test suites help to build trust between developers
  – 100% passing test suites help to avoid unnecessary communication
  – 100% passing test suites help to avoid synchronization points
Specific Areas where Trilinos Needs Improvements

- Reduce compilation times to speed rebuilds
  - Take all standard C++ headers out of Package_ConfigDefs.hpp and use only where needed
  - Make greater use of forward class declarations
  - Take greater advantage of the pImpl idiom for many more classes
  - Use explicit instantiation for as much templated C++ code as we can (i.e. templated on Scalar)
  - Exploit shared libraries (with Cmake build system)

- Create different categories of tests that get built and run for different purposes:
  - “Unit” tests (i.e. TDD tests)
  - “Basic integration” tests (i.e. pre-checkin tests)
  - “Regression” tests (i.e. nightly tests)
  - “Performance” tests
  - “Scalability” tests
  - “User-like” tests (i.e. backward compatibility tests)

- Improve installation testing:
  - Configure, build and install Trilinos, then reconfigure Trilinos tests and examples to build against installed Trilinos headers

- Improve the exception safety of our C++ codes (See Item 29 “Strive for exception-safe code” in “Effective C++ 3rd Edition”)

- Improve backward compatibility (tools, processes, policies, testing, ...)

- Nightly testing on more platforms (i.e. SCICO LAN Linux compilers, Sun, ...)

- Improve release process
Trilinos Release Process Improvements

- **Things to do before the branch for the release is created:**
  - Implement all functionality for the upcoming release
  - Keep all documentation and examples for “Stable” code up to date after each change
  - Put all “Experimental” code behind ifdefs so that it will not be included in the next release.
  - The “Stable” code for each package should almost always be in a releasable state
    => No reason to branch or tag individual packages separately before release branch creation
  - “Stable” code produces clean tests on all of the test platforms well before the release date
  - Perform at least one round of ports and acceptance tests with Trilinos Dev against all major customer platforms and applications a few weeks before the targeted release branch date.

- **Things to do after the branch for the release is created:**
  - Run automated scripts to automatically strip out all “Experimental” code and tests (This removes a lot of the need for complex tarball logic).
  - No changes are made to the branch except what are absolutely necessary to address serious defects.
  - Do (what should be) a final round of ports and acceptance tests against all major customer platforms and applications.
    - Resolve any new problems that have come up since the previous round of ports and tests conducted a few weeks prior. (Experience with SIERRA + Trilinos Integration shows that very few new issues come up.)
  - Change the version numbers inside of Trilinos. (NOTE: We need a more automated and uniform way of updating version numbers.)
  - Create the final tag.
  - Release the code.
Summary

• Partitioning of “Stable” and “Experimental” Code:
  – “Stable” Code and Tests:
    • Represents an important capability being used by an existing customer
    • Expected to be kept working at all times
  – “Experimental” Code and Tests:
    • All remaining code that is not “Stable” code
    • Represent fundamental research with informal low-quality software practices

• Goals for “Stable” and “Experimental” Code partitioning:
  – Allow crazy and impulsive algorithms research with “Experimental” Code
    • Benefit from ready to use “Stable” building blocks
    • Take advantage of everything the Trilinos environment has to offer
  – Maintaining “Stable” core allows:
    • Other “Experimental” research efforts can remain highly productive because their foundation is not constantly breaking
    • New requirements from “Stable” code needed to drive “Experimental” research code development can be rapidly developed and integrated in real time
  – Partitioning off “Experimental” code from “Stable” code
    • Avoid the problem of a top-heavy overly strict environment which does not allow for rapid research investigations.
  – By keeping “Stable” code in a near releasable state, we allow for fast and frequent releases of Trilinos.

We can have our cake and eat it too!