

Trilinos Checkin Testing of Primary Stable Code

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 “Stable” vs “Experimental” Code: Defined

- “Stable” Code and Tests:
 - “Meets one or more of the following criteria:
 - Represents an important capability being used by an existing, or
 - Represents a new capability that the authors are willing to stand behind
 - Does not mean it is being targeted for the next release
 - Expected to be kept working at all times on the primary development platform
 - Developed and maintained to be highly portable
 - Maintained at the high quality as defined by modern SE 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 macros that must be defined in order to be built.

Trilinos “Primary Stable” vs “Secondary Stable” Code

- Sub-categorizations of “stable” code:
 - “Primary Stable” code is “Stable” code that only depends on:
 - C, and C++ compilers
 - Fortran 77 compiler (optional)
 - BLAS and LAPACK
 - MPI
 - “Secondary Stable” code
 - Has 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)
 - Or, could be considered “Primary Stable” Code but is excluded from pre-checkin testing
 - Didasko
 - NewPackage
 - ...
- “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 extra configure-time options.

Stable (Primary and Secondary) and Experimental Code

- **Primary Stable Code and Tests:**

- All affected code should be built and tested *before* a checkin
- CATEGORY in `cmake/Trilinos[Packages,TPLs].cmake` set to “PS”
- Required TPL dependencies on **BLAS**, **LAPACK**, and **MPI** (or less)
- Configured with:
 - D Trilinos_ENABLE_ALL_PACKAGES:BOOL=ON \
 - D Trilinos_ENABLE_TESTS:BOOL=ON

- **Secondary Stable Code and Tests:**

- Represents an important (released) capability but has extra TPL dependencies
- *Note* be enabled for pre-checkin testing
- Tested by central framework resources (nightly integration testing)
- CATEGORY in `cmake/Trilinos[Packages,TPLs].cmake` set to “SS”
- Requires explicitly enabling “Stable” optional TPL dependencies
- Configured with:
 - D Trilinos_ENABLE_ALL_PACKAGES:BOOL=ON \
 - D Trilinos_ENABLE_SECONDARY_STABLE_CODE=ON \
 - D Trilinos_ENABLE_TESTS:BOOL=ON

- **Tertiary Stable Code and Tests?** (Right now just TPLs)

- **Experimental Code:**

- CATEGORY in `cmake/Trilinos[Packages,TPLs].cmake` set to “EX”
- Requires explicit enabling
- Tested by individual package teams (but posts results to main CDash dashboard)

Improving Stability of “Stable” code: Motivation

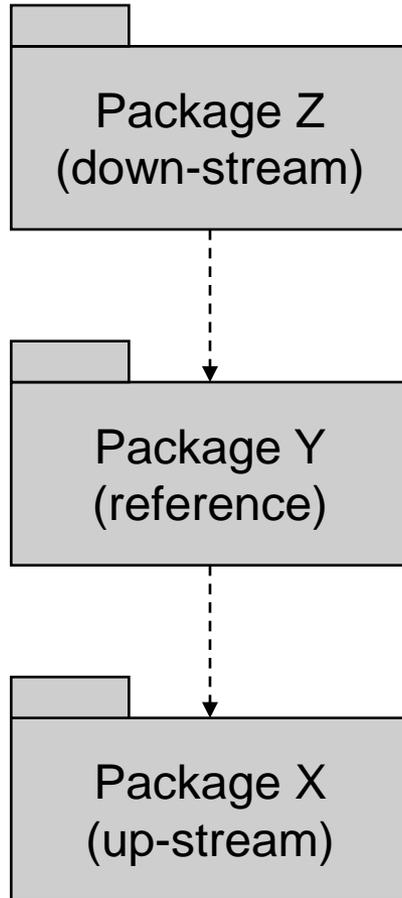
- Support deep stacks of vertically integrated Trilinos packages with production APPs
- Support tighter coupling and co-development with production APPs
 - SIERRA toolkit packages (STK_Mesh, STK_IO, ...)
 - Replace SIERRA framework code with Trilinos code (Teuchos::ParameterList, ...)
 - Many many others ...
- Support more frequent, safer, higher quality, lower risk releases of Trilinos
- Improve overall development productivity and software quality

See:

[Trilinos/doc/DevGuide/TrilinosSoftwareEngineeringImprovements/*.tex](#)

“Stable” Code: 100% Passing Test Policy

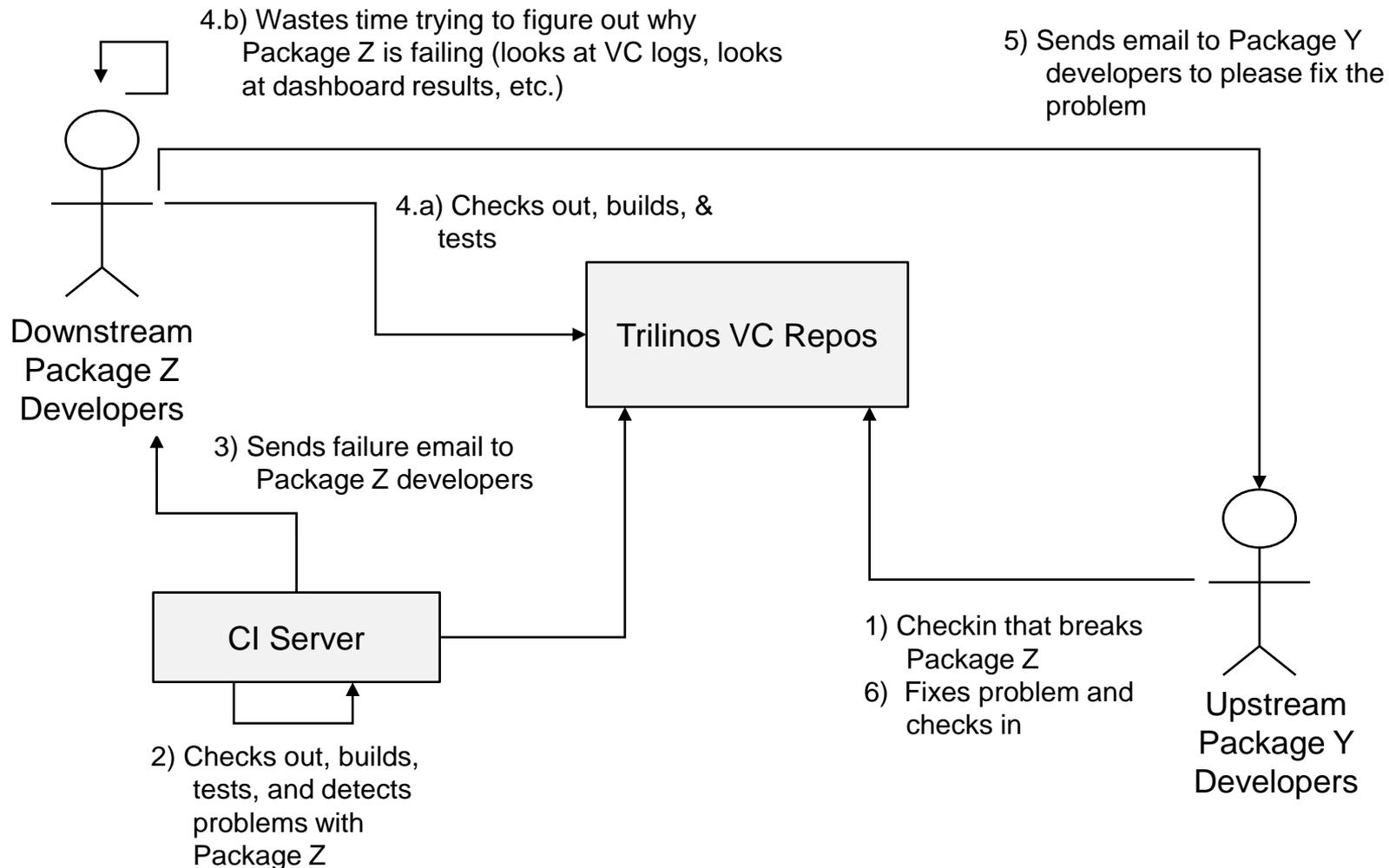
- All “Stable” code should have 100% passing tests 100% of the time on the primary development platforms as the norm instead of the exception.
- Achieving 100% passing tests on auxiliary development platforms is also a priority but is done in a secondary development loop.
- A failing test on any testing 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 loosening 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



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 is a 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

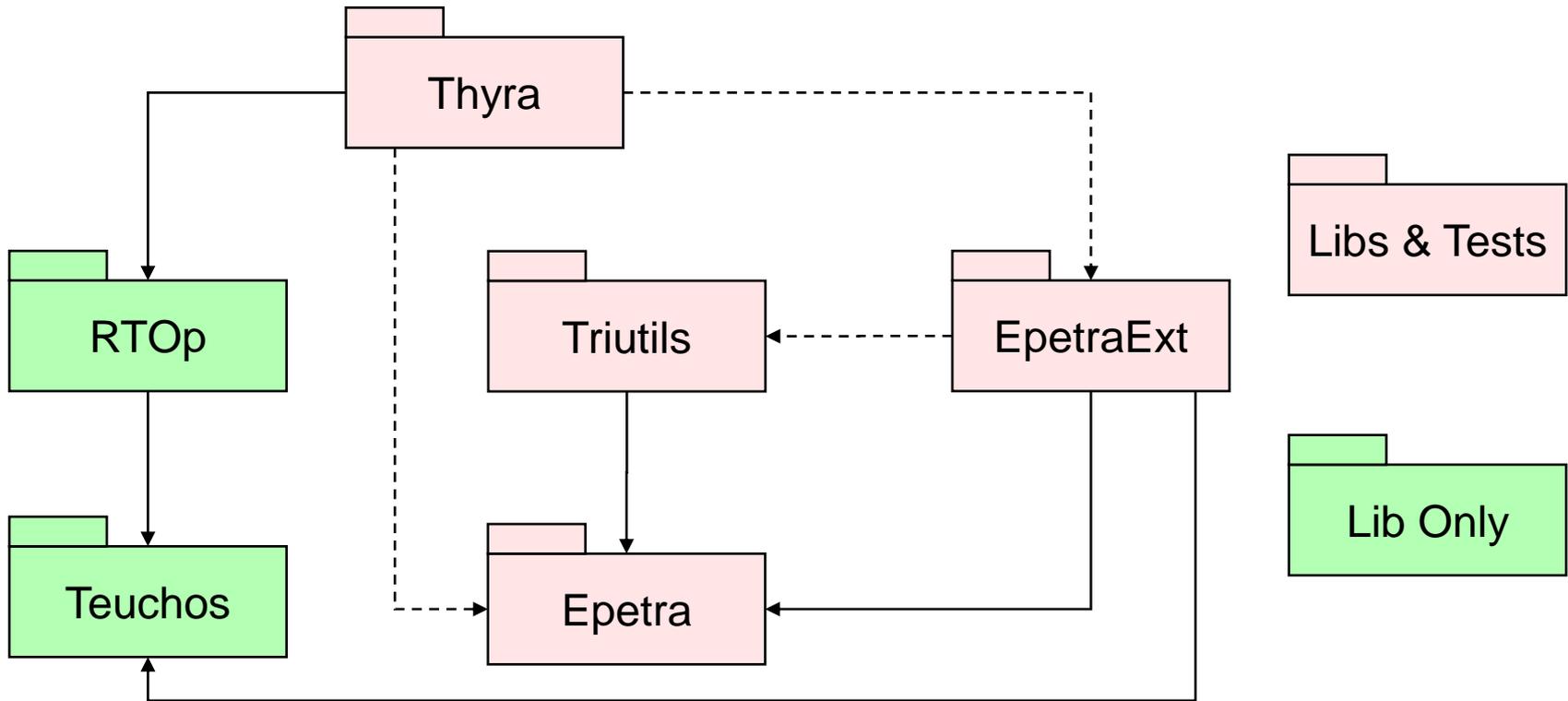
Waste Created By Lack of Sufficient Pre-Checkin Testing



- 90% of these problems can be avoided with sufficient pre-checkin testing!
- Catching the problem before checking in saves everyone wasted time!

Automatic Dependency Handling for Pre-Checkin Testing

```
$ cmake \  
  -D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \  
  -D Trilinos_ENABLE_Epetra:BOOL=ON \  
  -D Trilinos_ENABLE_ALL_FORWARD_DEP_PACKAGES:BOOL=ON \  
  -D Trilinos_ENABLE_TESTS:BOOL=ON \  
  ..
```



Pre-Checkin Testing: The checkin-test.py script

Python script that performs safe pre-checkin testing:

```
$ cd $TRILINOS_HOME
```

```
$ mkdir CHECKIN; echo CHECKIN >> .git/info/exclude; cd CHECKIN
```

```
$ ../checkin-test.py --do-all
```

- Automatically figures out what Trilinos packages have been changed
- Automatically enables all downstream packages
- Configures, builds and runs tests
 - Built-in Configurations:
 - MPI_DEBUG (Optimized compiler options, checked STL, etc.) (Do at least this build!)
 - SERIAL_RELEASE (varies other configure options)
 - Only enables Primary Stable Code!
 - Strong warning options (warnings as errors is a problem)
- Sends emails after each build case is finished
- Sends final email if it is okay to commit or not
- Can automatically do the commit at the end (Recommended)
- Fully customizable (enabled packages, build cases, etc.)
- Documentation: `checkin-test.py --help`

checkin-test.py: Example Driver Script

Script I used on my machine (checkin-test-<mymachine>.sh):

```
#!/bin/bash
EXTRA_ARGS=$@
echo "-DBUILD_SHARED_LIBS:BOOL=ON" > COMMON.config

/home/rabartl/PROJECTS/Trilinos.base/Trilinos/checkin-test.py \
--make-options="-j4" \
--ctest-options="-j4" \
--ctest-time-out=180 \
--commit-msg-header-file=checkin_message \
$EXTRA_ARGS
```

Run as (after symbolically linking into CHECKIN directory):

```
$ ./checkin-test-<mymachine>.sh -do-all -commit
```

Example driver scripts (I symbolically link these):

```
sampleScripts/checkin-test-cygwin-rabartl.sh
sampleScripts/checkin-test-<mymachine>.sh
sampleScripts/checkin-test-scicolan-rabartl.sh
...
```



checkin-test.py: Recommended Workflow

A) Fill out the checkin checklist message in a temporary text file
'checkin_message'

B) Do local git commits (optional)

C) Run the checkin-test.py script:

```
$ ./checkin-test-mymachine.sh --do-all [--commit ...]
```

D) Go do something useful (e.g. **go home**, check email, review a paper, work on a paper, talk with someone, ..)

D) Check your email later to see what happens

Consequences:

- Documents a bullet-proof process for configuring, building, and testing Trilinos
 - Does the VC commands to do a safe global checkin (ease git transition)
 - Enjoy fewer bad checkins
 - Spend less time driving the checkin process
- 

Directory Structure for auto-generated log files

CHECKIN/

checkin-test.out

commitFinal.out

commitInitial.out

pullInitial.out

push.out

...

MPI_DEBUG/

configure.out

make.out

ctest.out

...

SERIAL_RELEASE/

...

See log files while configure, build, or test is being run do, for example:

```
$ tail -f MPI_DEBUG/make.out
```

checkin-test.py: Cost of Pre-Checkin Testing (Average Case)

A) Enabling just ML and tests/examples in downstream packages

Enabled packages (libraries) (29/52): Teuchos, RTOp, Kokkos, Epetra, Zoltan, Shards, Triutils, Tpetra, EpetraExt, Thyra, Isorropia, AztecOO, Galeri, Amesos, Pamgen, Ifpack, ML, Belos, Stratimikos, Meros, FEI, Anasazi, , Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance

Enabled packages (tests/examples) (10/52): ML, Belos, Stratimikos, Meros, FEI, NOX, Moertel, Rythmos, MOOCHO, Sundance

<fast-machine>, shared libs, from scratch

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	24.2	3.9	438
SERIAL_RELEASE	18.1	1.1	426

<fast-machine>, shared libs, rebuildid

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	0.7	4.0	438
SERIAL_RELEASE	0.4	1.2	426

<average-machine>, shared libs, from scratch

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	59.0	6.5	434
SERIAL_RELEASE*	30.4	1.3	350

<average-machine>, shared libs, rebuildid

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	1.4	6.6	434
SERIAL_RELEASE*	0.7	1.3	350

- **With shared libraries, rebuilds can be very fast!**
- **Use a fast machine to checkin from!**

* Sundance disabled on <average-machine> for serial build (see bug ???)

checkin-test.py: Cost of Pre-Checkin Testing (Worst Case)

B) Enabling Teuchos and tests/examples in downstream packages

Enabled packages (libraries) (34/52): Teuchos, RTOp, Kokkos, Epetra, Zoltan, Shards, GlobiPack, Triutils, Tpetra, EpetraExt, Thyra, OptiPack, Isorropia, AztecOO, Galeri, Amesos, Pamgen, Ipack, Komplex, ML, Belos, Stratimikos, Meros, FEI, Anasazi, RBGen, Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance, CTrilinos

Enabled packages (tests/examples) (22/52): Teuchos, OptiPack, Isorropia, AztecOO, Galeri, Amesos, Ipack, Komplex, ML, Belos, Stratimikos, Meros, FEI, Anasazi, RBGen, Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance

<fast-machine>, shared libs, from scratch

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	48.0	8.34	1140
SERIAL_RELEASE	37.3	1.9	1147

<fast-machine>, shared libs, rebuild

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	1.1	8.1	1140
SERIAL_RELEASE	1.2	2.1	1147

<average-machine>, shared libs, from scratch

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	103.0	12.0	1136
SERIAL_RELEASE*	63.5	2.5	1071

<average-machine>, shared libs, rebuild

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	2.3	12.0	1136
SERIAL_RELEASE*	1.49	2.5	1071

* Sundance disabled on <average-machine> (see bug ???)

checkin-test.py: Shared Libraries vs. Static Libraries

B) Enabling Teuchos and tests/examples in downstream packages

Enabled packages (libraries) (34/52): Teuchos, RTOp, Kokkos, Epetra, Zoltan, Shards, GlobiPack, Triutils, Tpetra, EpetraExt, Thyra, OptiPack, Isorropia, AztecOO, Galeri, Amesos, Pamgen, Ipack, Komplex, ML, Belos, Stratimikos, Meros, FEI, Anasazi, RBGen, Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance, CTrilinos

Enabled packages (tests/examples) (22/52): Teuchos, OptiPack, Isorropia, AztecOO, Galeri, Amesos, Ipack, Komplex, ML, Belos, Stratimikos, Meros, FEI, Anasazi, RBGen, Sacado, Intrepid, NOX, Moertel, Rythmos, MOOCHO, Sundance

<average-machine>, shared libs, from scratch

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	103.0	12.0	1136
SERIAL_RELEASE*	63.5	2.5	1071

<average-machine>, shared libs, rebuild

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	2.3	12.0	1136
SERIAL_RELEASE*	1.49	2.5	1071

<average-machine>, static libs, from scratch

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	115.3	10.7	1136
SERIAL_RELEASE*	72.4	2.7	1071

<average-machine>, static libs, rebuild

Build Type	Build (min)	Test (min)	#tests
MPI_DEBUG	18.9	10.4	1136
SERIAL_RELEASE*	6.6	2.4	1071

- **Rebuilds with shared libs can be *much* faster that with static libs!**

* Sundance disabled on <average-machine> for serial build (see bug ???)



Speeding up Pre-Checkin Testing: Current Approaches

- 100% safe approaches:

- Checkin from a fast workstation no matter where you develop (easy with git)
- Keep private development and checkin builds separate
- Enabled shared libraries (`-DBUILD_SHARED_LIBS:BOOL=ON`)
- Keep the CHECKIN builds up to date (could use crontab or just manually)

- Less than 100% safe approaches (from better to worst):

- Do only MPI_DEBUG build (`--without-serial-release`)
- Disallow enabling all packages (`--enable-all-packages=off`)
 - Example: Disables enabling all packages when `cmake/TrilinosPackages.cmake` changes
- Disable forward packages (`--no-enable-fwd-packages`)
 - Example: Only tests in the package have changed
 - Example: Good unit tests and minimal changes
- Disabling specific downstream packages (`--disable-packages=P1,...`)
 - Example: Disabling Sundance when testing Tpetra
- Enabling only specific packages (`--enable-packages=P1,...`)
 - Example: Only test a few packages

`--enable-all-packages=off --enable-packages=Tpetra,Belos,Anasazi`

Improving Pre-Checkin Testing: Future Approaches

- Speeding up pre-checkin testing:
 - Move to explicit template instantiation
 - Forward declarations
 - Use plmpl idiom (faster rebuilds)
 - Remove standard C++ headers out of Package_ConfigDefs.hpp
 - Trim down number of “Basic Integration” test executables
 - More unit tests, faster more minimal basic integration tests
 - Move to a sub-package architecture in the CMake build system
- Improving consistency of pre-checkin testing:
 - Standardize versions of GCC, MPI, BLAS, LAPACK etc. ...
=> Official Trilinos Developers Toolset
- Improving the portability testing of pre-checkin testing:
 - Strong warnings and warnings as errors
 - Requires standard versions of GCC and MPI!
=> Official Trilinos Developers Toolset

Extra Build/Test Cases

- Motivation:

- Your development work involves working with Secondary Stable or Experimental code and you want to combine this with other standard builds in the same process.

- Allow for extra user-defined build cases:

- `-- extra-builds=BUILD1,BUILD2,...,BUILDN`

- Example: Test Secondary Stable Code and TPLs

- `$ echo "-DTPL_ENABLE_SCOTCH:BOOL=ON" >> WITH_SCOTCH.config`

- `$./checkin-test-mymachine.sh --extra-builds=WITH_SCOTCH --do-all`

Pre-Checkin Testing: Summary

- Using this script will improve the stability of Trilinos for everyone involved!
- Bad reasons to do a sloppy checkin:
 - “I want to integrate my code frequently”
 - => Good motivation but not as important good testing
 - => Checking in once a day is usually sufficient
 - “I need to get this revision to a collaborator ASAP”
 - => Just have them pull directly from your local git repository
 - “In am doing porting work and can’t afford a complete test on the machine”
 - => Pull local commits back to your git local working directory your workstation and commit from there (remote test/push)
 - “I am in a good point to checkpoint my changes”
 - => Do a local git commit
 - “I want to backup my work with history”
 - => Use git to publish to a “backup” repository on a different machine
 - “I want to checkin to feel a sense of completion”
 - => Mental problem, seek help
- Please read “checkin-test.py –help” and give this a try!
- Please ask questions, give feedback!