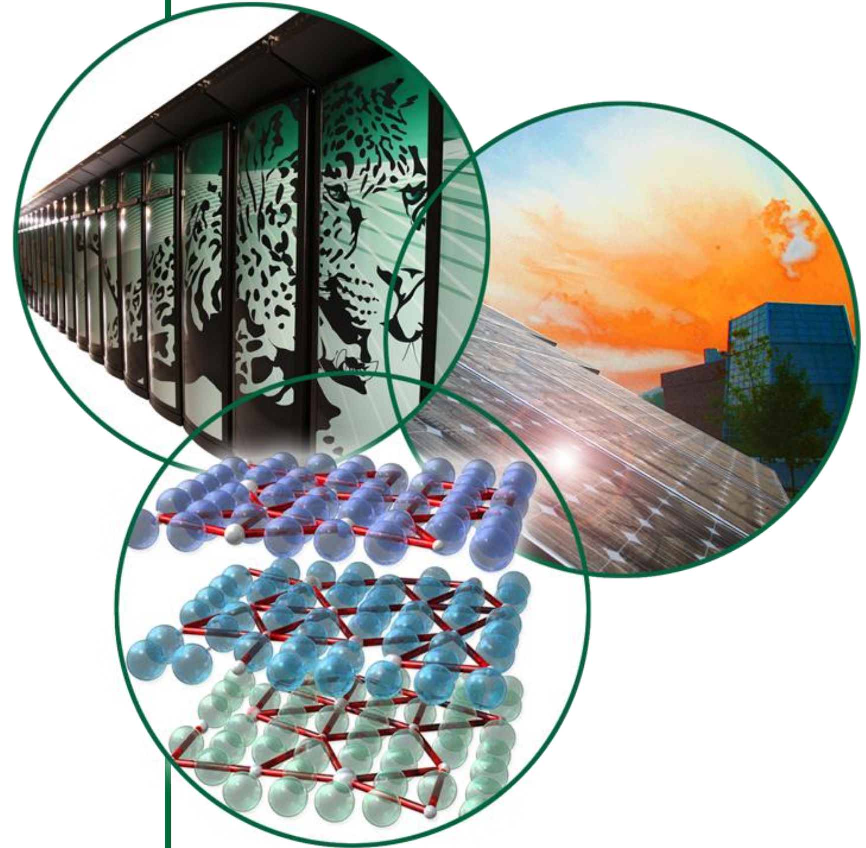
Trilinos Adoption of the TriBITS Lifecycle Model

Trilinos Users Group Meeting

Developers Day

November 1, 2012



Roscoe A. Bartlett

CASL Vertical Reactor Integration Software Engineering Lead

Trilinos Software Engineering Technologies and Integration Lead

Computer Science and Mathematics Div

U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY

MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Outline

- *Overview of the TriBITS Lifecycle Model*
- *Trilinos Spring Developers Meeting 2012 Software Survey*
- *Ideas for the Adoption of the TriBITS Lifecycle Model*
- *Recommended Stages for the Adoption of the TriBITS Lifecycle Model*
- *Summary and Open Discussion*

TriBITS Lifecycle Model Overview

OAK RIDGE
National Laboratory

TriBITS Lifecycle Model 1.0 Document

SANDIA REPORT

SAND2012-0561
Unlimited Release
Printed February 2012

TriBITS Lifecycle Model

Version 1.0

A Lean/Agile Software Lifecycle Model for Research-based Computational Science and Engineering and Applied Mathematical Software

Roscoe A. Bartlett
Michael A. Heroux
James M. Willenbring

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

http://www.ornl.gov/~8vt/TribitsLifecycleModel_v1.0.pdf

Goals for the TriBITS Lifecycle Model

- ***Allow Exploratory Research to Remain Productive***: Only minimal practices for basic research in early phases
- ***Enable Reproducible Research***: Minimal software quality aspects needed for producing credible research, researches will produce better research that will stand a better chance of being published in quality journals that require reproducible research.
- ***Improve Overall Development Productivity***: Focus on the right SE practices at the right times, and the right priorities for a given phase/maturity level, developers work more productively with acceptable overhead.
- ***Improve Production Software Quality***: Focus on foundational issues first in early-phase development, higher-quality software will be produced as other elements of software quality are added.
- ***Better Communicate Maturity Levels with Customers***: Clearly define maturity levels so customers and stakeholders will have the right expectations.

Self-Sustaining Software: Defined

- ***Open-source***: The software has a sufficiently loose open-source license allowing the source code to be arbitrarily modified and used and reused in a variety of contexts (including unrestricted usage in commercial codes).
- ***Core domain distillation document***: The software is accompanied with a short focused high-level document describing the purpose of the software and its core domain model.
- ***Exceptionally well testing***: The current functionality of the software and its behavior is rigorously defined and protected with strong automated unit and verification tests.
- ***Clean structure and code***: The internal code structure and interfaces are clean and consistent.
- ***Minimal controlled internal and external dependencies***: The software has well structured internal dependencies and minimal external upstream software dependencies and those dependencies are carefully managed.
- ***Properties apply recursively to upstream software***: All of the dependent external upstream software are also themselves self-sustaining software.
- ***All properties are preserved under maintenance***: All maintenance of the software preserves all of these properties of self-sustaining software (by applying Agile/Emergent Design and Continuous Refactoring and other good Lean/Agile software development practices).

TriBITS Lifecycle Maturity Levels

-1: Unspecified Maturity (UM) Code

- Provides no official indication of maturity or quality
- i.e. “Opt Out” of the TriBITS Lifecycle Model

0: Exploratory (EP) Code

- Primary purpose is to explore alternative approaches and prototypes, not to create software.
- Does not provide sufficient unit (or otherwise) testing to demonstrate correctness.
- Often has a messy design and code base.
- Does not provide a direct foundation for creating production-quality code.

1: Research Stable (RS) Code

- Strong unit and verification tests are written as the code/algorithms are being developed
- Has a very clean design and code base maintained through emergent design and constant refactoring.
- Generally does not have higher-quality documentation, user input checking and feedback, space/time performance, portability, backward compatibility, or acceptance testing.

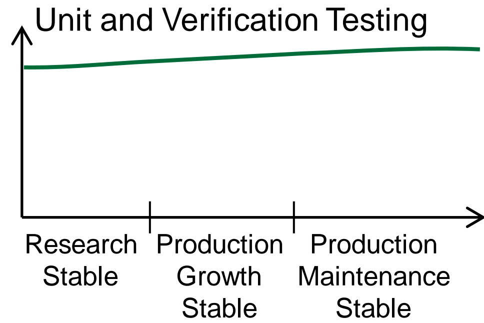
2: Production Growth (PG) Code

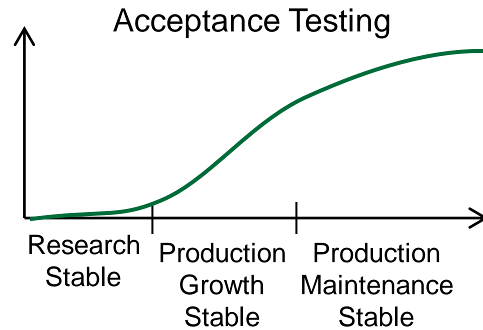
- Provides increasingly improved checking of user input errors and better error reporting, documentation/examples/tutorials, portability, space/time performance.
- Maintains clean structure through constant refactoring.
- Maintains increasingly better regulated backward compatibility.
- Has expanding usage in more customer codes.

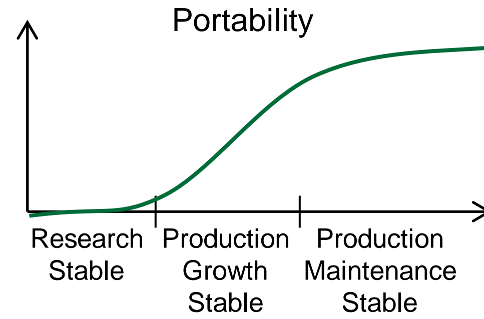
3: Production Maintenance (PM) Code

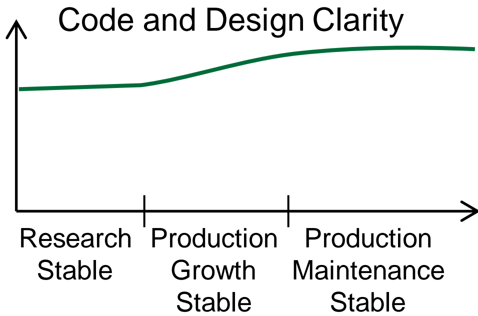
- Primary development includes mostly just bug fixes and performance tweaks.
- Maintains rigorous backward compatibility with few/none deprecated features or breaks in backward compatibility.
- Could be maintained by parts of the user community if necessary (i.e. as Self Sustaining Software).

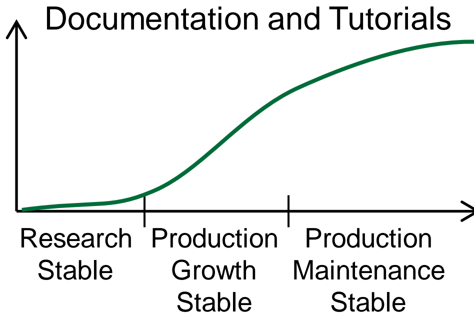
Proposed TriBITS Lean/Agile Lifecycle

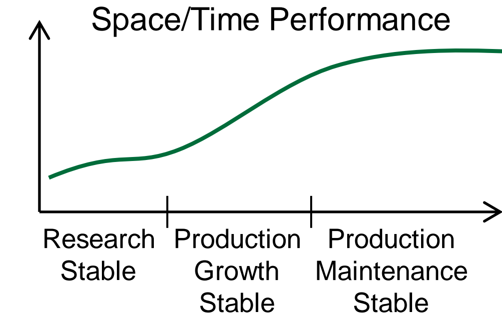


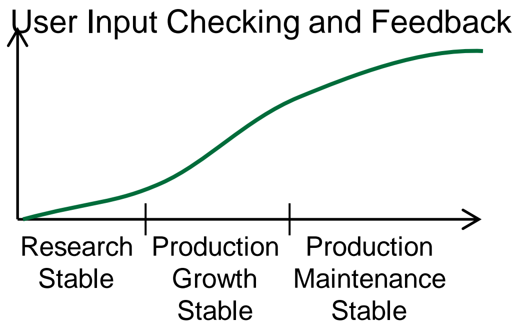


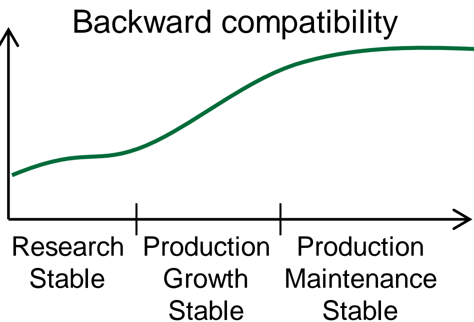


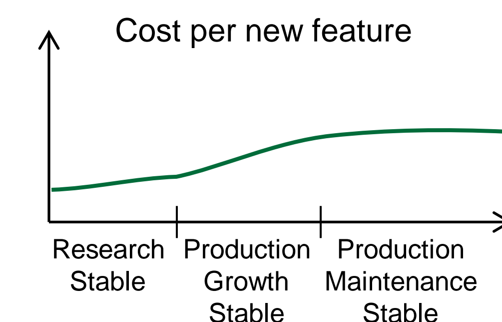






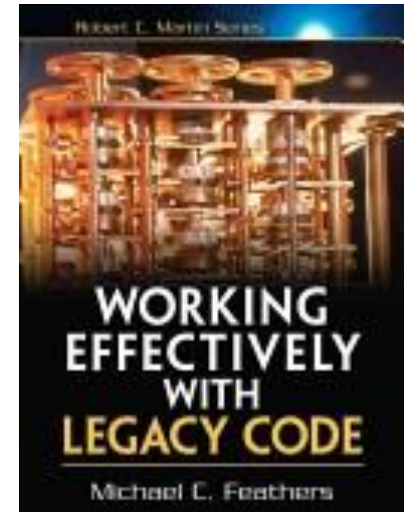






Time

Grandfathering of Existing Packages

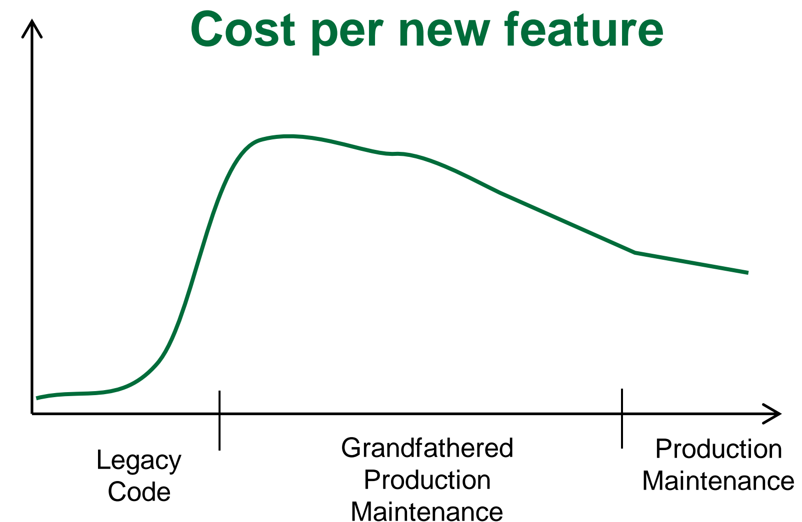


Agile Legacy Software Change Algorithm:

1. Cover code to be changed with tests to protect existing behavior
2. Change code and add new tests to define and protect new behavior
3. Refactor and clean up code to well match current functionality

Grandfathered Lifecycle Phases:

1. Grandfathered Research Stable (GRS) Code
2. Grandfathered Production Growth (GPG) Code
3. Grandfathered Production Maintenance (GPM) Code



NOTE: After enough iterations of the Legacy Software Change Algorithm the software may approach Self-Sustaining software and be able to remove the "Grandfathered" prefix!

The Legacy Software Change Algorithm is the Lynch Pin of the TriBITS Lifecycle Model!

How is this different from what is typically done?

Validation-Centric Approach (VCA): Common Lifecycle Model for CSE Software

Central elements of validation-centric approach (VCA) lifecycle model

- Develop the software by testing against real early-adopter customer applications
- Manually verify the behavior against applications or other test cases

Advantages of the VCA lifecycle model:

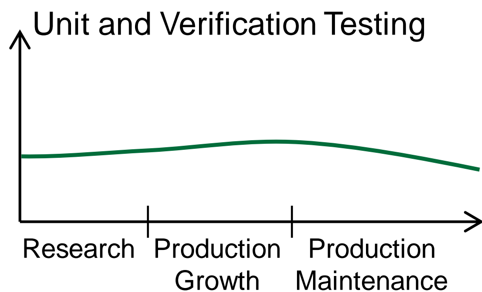
- Assuming customer validation of code is easy (i.e. linear or nonlinear algebraic equation solvers => compute the residual) …
- Can be very fast to initially create new code
- Works for the customers code right away

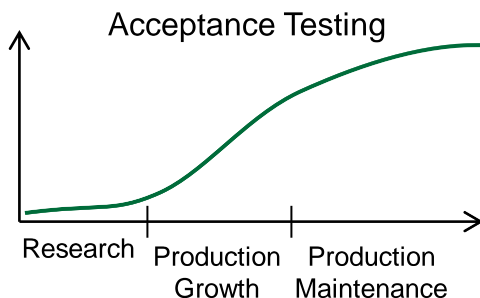
Problems with the VCA lifecycle model:

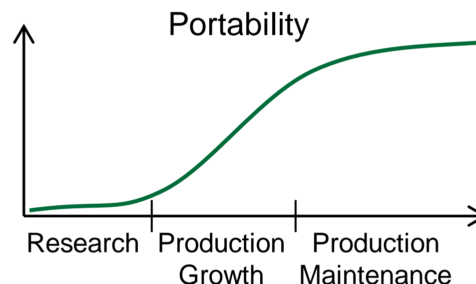
- Does now work well when validation is hard (i.e. ODE/DAE solvers where no easy to compute global measure of error exists)
- Re-validating against existing customer codes is expensive and is often lost (i.e. the customer code becomes unavailable).
- Difficult and expensive to refactor: Re-running customer validation tests is too expensive or such tests are too fragile or inflexible (e.g. binary compatibility tests)

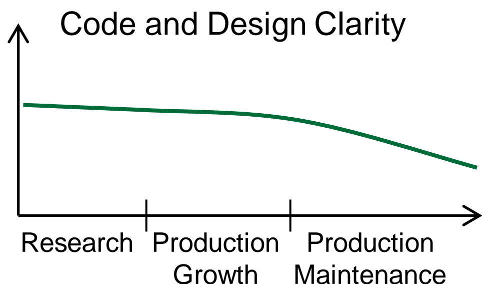
VCA lifecycle model often leads to unmaintainable codes that are later abandoned!

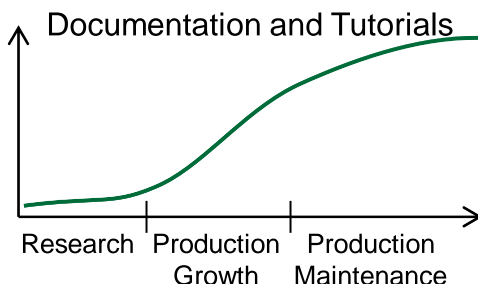
Typical non-Agile (i.e. VCA) CSE Lifecycle

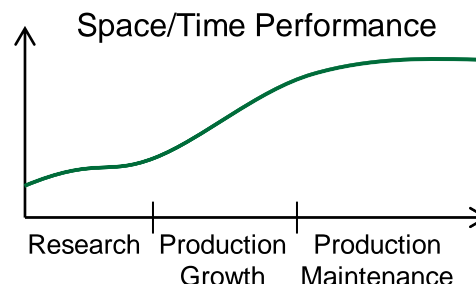


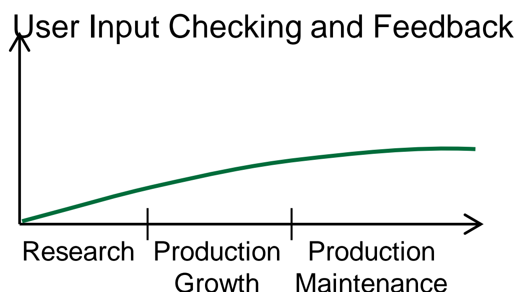


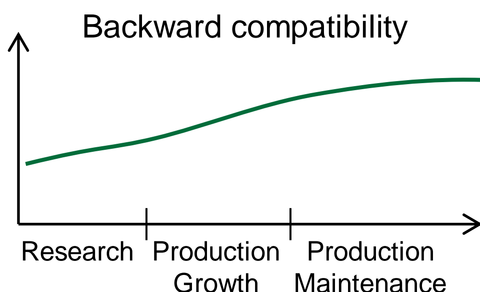


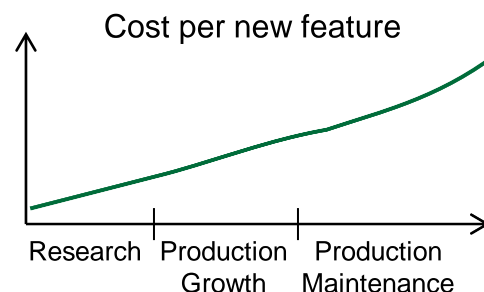












Trilinos Spring Developers Meeting 2012 Survey

OAK RIDGE National Laboratory

Top committers to Trilinos 11/2011 - 2012

$ eg shortlog -ns --after="10/29/2011" --before="10/30/2012“ # Note duplicates!

637 Mark Hoemmen
540 Roscoe A. Bartlett
534 Jeremie Gaidamour
503 Lee Ann Riesen
364 Tobias Wiesner
310 Jonathan Hu
308 Eric C. Cyr
288 Todd Kordenbrock
249 K Devine
216 Stephen Kennon
203 Greg Sjaardema
195 Christopher G. Baker
189 Roger Pawlowski
171 Brent Perschbacher
167 Chris Siefert
128 Carter Edwards
126 Eric Phipps
105 Mehmet Deveci
96 Alan Williams
90 James M. Willenbring
84 Kevin Long
78 Erik Boman
75 Will Dicharry

70 Siva Rajamanickam
63 bktidwe
44 K. Devine
36 James Foucar
35 Heidi Thornquist
32 Ron Oldfield
31 H. Carter Edwards
31 raoldfi
29 Karla
27 Kurtis Nusbaum
25 Ben Seefeldt
25 Boyd Tidwell
25 Chetan Jhurani
25 Matt Bettencourt
25 jhurani@txcorp.com
24 Julien Cortial
24 Kendall Hugh Pierson
21 Mike Heroux
21 William F. Spotz
17 Andy Salinger
16 Dan Sunderland
16 rouson
14 Karla Morris

14 Karla Morris
14 Robert C. Kirby
12 Ray Tuminaro
11 Nico Schlömer
10 James Willenbring
10 Kevin Deweese
10 Todd S. Coffey
10 Travis Austin
9 Brian J. Miller
8 Curtis C. Ober
8 Damian Rouson
8 David Day
7 Heidi K. Thornquist
7 Michael Parks
7 Nathan Crane
6 David Hensinger
5 Bill Spotz
5 Brian Adams
5 Nate Roehrig
5 Radu Popescu
5 Vitus Leung
…
4 and less commits …

About 60 unique committers to Trilinos over last year with >= 5 commits

Trilinos Spring Dev Meeting 2012 Survey

- **View questions and results:** https://docs.google.com/spreadsheet/gform?key=0AoqqHXGhvq-VdHVOVzI5bHFaM2ZCYWFTNlo0OEFrWUE&gridId=0#chart
- **Analysis of results:** 2012 Trilinos Spring Dev Meeting Survey
- **Summary of significant results:**
  - Smaller sample: **22 total responses** out of **≈ 60 Trilinos developers** making >= 5 commits 10/29/2011 to 10/30/2012. **What about these other 35+ developers?**
  - On a scale from 1 to 10, how important do you believe it is for the Trilinos team to achieve a significantly higher level of software quality within the next 3 years? => **Average = 7.2**
  - On a scale from 1 to 10, how important do you believe it is for Trilinos to adopt a better-defined software lifecycle model? => **Average = 6.3**
  - Do you support an effort to adapt and adopt the TriBITS Lifecycle Model for Trilinos? => **Yes/No = 6.3**
  - Would you be willing to commit to using the Legacy Software Change Algorithm so that your Trilinos code could be grandfathered into a new Trilinos Lifecycle Model based on the TriBITS Lifecycle Model? => **Yes/No = 3.7, Undecided/Total = 0.36.**
- **Final conclusions from the survey:**
  - **Strong sense of need and support for adopting the TriBITS Lifecycle Model by those who read all or some of the TriBITS Lifecycle Model document.**
  - **No-one who read the entire TriBITS Lifecycle Model document opposed its adoption.**
  - **Medium support for committing to the Legacy Software Change Algorithm to Grandfather existing Trilinos packages.**
  - **Over 1/3 of respondents were undecided about Legacy Software Change Algorithm => Improve with Training and Education?**

Ideas for the Adoption of the TriBITS Lifecycle Model by Trilinos

OAK RIDGE National Laboratory

Overview of Implementation Levels/Options?

A) Minimum Level of Adoption:

- Clearly publically document the TriBITS Lifecycle Model maturity levels and train Trilinos developers in essentials.
- Use vocabulary and terminology of the TriBITS Lifecycle Model in all communication (developers and users).
- Label existing and new Trilinos packages (at subpackage level) as EP, (G)RS, (G)PG, (G)PM, UM … Display on web pages, in TriBITS system, etc.

B) Medium Level of Adoption:

- Implement and institute voluntary training of SE practices for existing and new Trilinos developers.
- Use maturity levels to determine amount and type of testing (e.g. backward compatibility testing, coverage testing, valgrind testing, etc.) for various maturity levels.
- Define software quality metrics for testing, code clarity and quality, user input checking and error feedback, portability, space/time performance, documentation/examples/tutorials, etc.
- Define targets for software quality metrics for specific maturity levels.

C) Fuller Level of Adoption:

- Make training of SE practices mandatory for existing and new Trilinos developers before being allowed to change and commit to non-EP non-UM packages.
- Define and mandate software engineering standards/practices for higher maturity/criticality packages (e.g. for (G)PM code: code reviews for all commits to, tests for all code changed and added, bug/ticket for all changes, etc.).
- Determine, track (keep up to date) and publish software quality metrics evaluations for packages, subpackages, capabilities, to improve communication (developers and users) and to foster improvement.

Specific Infrastructure Changes?

Changes to TriBITS System?

- Change exist testing-based (not maturity-based) TriBITS CATEGTORIES property
  - from EX (Experimental), PS (Primary Stable), and SS (Secondary Stable)
  - to NT (Non Tested), PT (Primary Tested), and ST (Secondary Tested)
  - Advantages: Would confusion with maturity/quality levels
  - Disadvantages: Would break backward compatibility (or we could depreciate old names)
- Add a new field/property for a TriBITS (sub)packages MATURTY_LEVEL:
  - EP (Exploratory), Research Stable (RS), Production Growth (PG), Production Maintenance (PM), Grandfathered Research Stable (GRS), Grandfathered Production Growth (GPG), Grandfathered Production Maintenance (GPM)
- Enforce maturity level standards for required (sub)package dependencies:
  - Example: (G)PM (sub)packages can only have required dependencies on (G)PM (sub)packages
  - Example: (G)PG (sub)packages can only have required dependencies on (G)PG or (G)PM (sub)packages
- Allow (sub)packages to designate specific tests/examples to participate or *NOT* to participate in backward compatibility testing.
- Add checkin-test.py --maturity-level=<maturitylevel> to affect what downstream packages are enabled and/or tested.

Changes to Automated Testing?

- Perform backward compatibility testing by default on all tests/examples for (G)PM.
- Only perform backward compatibility testing on specific designated tests/examples for (G)PG.
- Perform no backward compatibility testing at all for (G)PS.
- Separate coverage and memory/valgrind testing into categories according to maturity level?

Specific Developer Processes & Standards?

- **All Maturity Level Code:**
  - Focus on improving properties of Self Sustaining Software.
  - Run checkin-test.py enabling on all downstream Primary Testing (PT) (G)PG and (G)PM packages.
- **(Grandfathered) Research Stable ((G)RS) Code:**
  - Require/expect near 100% line coverage for all new code and recently modified code (i.e. using the Legacy Software Change Algorithm)
  - Expect and strive for "clean" and "clear" code (see Code Complete 2nd Edition)
  - Expect running checkin-test.py script on all downstream (G)RS and above packages.
- **(Grandfathered) Production Growth ((G)PG) Code:**
  - Bug/ticket recommended be filed for every change that will affect interfaces or behavior.
  - Pre- or post-push documented code reviews required for all changes that affect interfaces or behavior.
  - Post-push code reviews and issue bug/ticket issue tracking are optional but useful for all changes.
  - Specifically mark subset of tests/examples as backwards compatible.
- **(Grandfathered) Production Maintenance ((G)PM) Code:**
  - Bug/ticket must be filed to justify and document every change.
  - Legacy software change algorithm must be applied to change non-tested code.
  - Code being changed or new code added should have near 100% line coverage (determined pre-push).
  - Code reviews are required before pushing to 'master' branch with reviews documented in bug/ticket.
  - Specifically mark subset of low-level unit tests as *NOT* backwards compatible.

Open Questions?

Key Questions:

- **How do we roll out the TriBITS Lifecycle Model?**
  => Start with a pilot project and apply first to a core set of Trilinos packages (e.g. Teuchos, Epetra, Kokkos, Tpetra, Thyra, Stratimikos, etc.) and corresponding development teams?
- **How do we adjust and modify the lifecycle model and processes?**
  => Use the pilot project package development teams to work out the bugs and make adjustments?
- **In the end, how do we deal with minority decent?**
  => Allow it and have others inconvenienced and compensate for it?
  => Force compliance for critical software?

Others Questions:

- **Experimenting in a (G)PG or (G)PM package?**
  - Short lived:
    => Work on a local branch and back up and share using an intermediate git repo?
  - Long lived:
    => Break off a new lower-maturity subpackage?
    => Use CMake conditionals and/or compile-time ifdefs to optionally enable/disable lower-maturity code/tests?

Recommended Stages for the Adoption of the TriBITS Lifecycle Model by Trilinos?

OAK RIDGE
National Laboratory

Recommended Stages/Actions?

- **Documenting and training for key elements of the TriBITS Lifecycle Model**
  - Concisely document the TriBITS Lifecycle Model on Trilinos websites
  - Train pilot-project package team members in the TriBITS Lifecycle Model, Self Sustaining Software, and maturity level definitions and issues … e.g. self study? presentation and workshop? Webinar?
- **Implement support for TriBITS Lifecycle Model in TriBITS**
  - MATURITY_LEVEL (sub)package property and dependency logic tests
  - Change CATEGORIES to TESTING_CATEGORIES and names EX, PS, SS to NT, PT, ST?
- **Assign existing core Trilinos packages to specific (grandfathered) maturity levels**
  - (G)PM: Teuchos, Epetra, AztecOO, NOX, …
  - (G)PG: Kokkos, Tpetra, Thyra, Stratimikos, …
  - Mark most other packages as UM (Unspecified Maturity).
- **Provide training for Legacy Software Change Algorithm to pilot team members**
  - Reading group for "Working Effectively with Legacy Code"?
  - Webinars?
  - One-day workshop?
- **Provide general software development training?**
  - Unit testing, test driven development, structured incremental refactoring, Agile-emergent design?
  - Code reviews?
- **Define standards/processes and quality metrics for different maturity levels**
  - See previous slides for examples

Summary and Open Discussion

Summary of TriBITS Lifecycle Model

- **Motivation:**
  - Allow Exploratory Research to Remain Productive
  - Enable Reproducible Research
  - Improve Overall Development Productivity
  - Improve Production Software Quality
  - Better Communicate Maturity Levels with Customers
- **Self Sustaining Software => The Goal of the Lifecycle Model**
  - Open-source
  - Core domain distillation document
  - Exceptionally well testing
  - Clean structure and code
  - Minimal controlled internal and external dependencies
  - Properties apply recursively to upstream software
  - All properties are preserved under maintenance
- **Lifecycle Phases:**
  - -1: Unspecified Maturity (UM) Code
  - 0: Exploratory (EP) Code
  - 1: Research Stable (RS) Code
  - 2: Production Growth (PG) Code
  - 3: Production Maintenance (PM) Code
- **Grandfathering existing Legacy packages into the lifecycle model:**
  - Apply Legacy Software Change Algorithm => Slowly becomes Self-Sustaining Software over time.
  - Add "Grandfathered" prefix to RS, PG, and PM phases.

Open Discussion: What Do You Think?

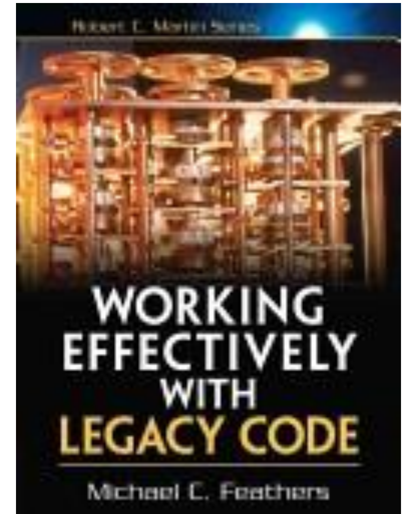
- ???

THE END

Key Agile Technical Concepts, Practices and Skills

Definition of Legacy Code and Changes

Legacy Code = Code Without Tests

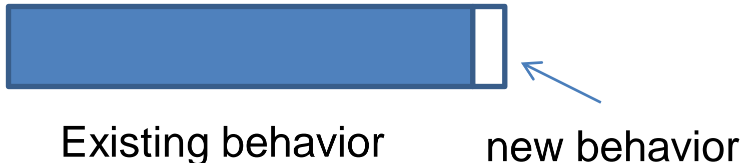


"Code without tests is bad code. It does not matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our ode is getting better or worse."

Source: M. Feathers. Preface of "Working Effectively with Legacy Code"

Reasons to change code:

- Adding a Feature
- Fixing a Bug
- Improving the Design (i.e. Refactoring)
- Optimizing Resource Usage



Preserving behavior under change:

"Behavior is the most important thing about software. It is what users depend on. Users like it when we add behavior (provided it is what they really wanted), but if we change or remove behavior they depend on (introduce bugs), they stop trusting us."

Source: M. Feathers. Chapter 1 of "Working Effectively with Legacy Code"

Key Agile Technical Practices

- **Unit Testing**
  - Re-build fast and run fast
  - Localize errors
  - Well supports continuous integration, TDD, etc.
- **System-Level Testing**
  - Tests on full system or larger integrated pieces
  - Slower to build and run
  - Generally does not well support CI or TDD.

Quick and Dirty Unit Tests

Courser-grained tests that are relatively fast to write but take slightly longer to rebuild and run than pure "unit tests" but cover behavior fairly well but don't localize errors as well as "unit tests".

- **(Unit or Acceptance) Test Driven Development (TDD)**
  - Write a compiling but failing (unit or system or acceptance) test and verify that it fails
  - Add/change minimal code until the test passes (keeping all other tests passing)
  - Refactor code to make more clear and remove duplication
  - Repeat (in many back-to-back cycles)
- **Incremental Structured Refactoring**
  - Make changes to restructure code without changing behavior (or performance, usually)
  - Separate refactoring changes from changes to change behavior
- **Agile-Emergent Design**
  - Keep the design simple and obvious for the current set of features (not some imagined set of future features)
  - Continuously refactor code as design changes to match current feature set

These are real skills that take time and practice to acquire!

Quick and Dirty Unit Tests vs. Manual Tests

Quick and Dirty Unit Tests

- Courser-grained tests that instantiate several objects and/or execute several functions before checking final results.
- Courser-grained than “unit tests” but much finer grained than “system tests”
- May be written much more quickly than pure “unit tests” but still cover almost the same behavior.
- Might be slightly slower to rebuild and run than pure “unit tests”
- May not localize errors as well as pure “unit tests”
- Can be later broken down into finer-grained unit tests if justified

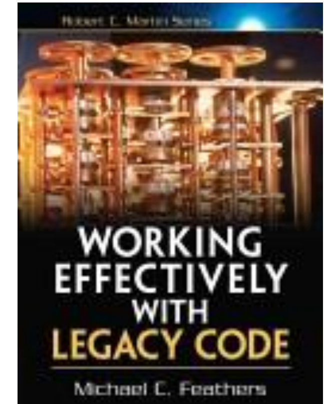
Quick and Dirty Unit Testing vs. Manual Verification Tests

- Manual verification tests take longer to perform while doing development than you may realize.
- Manual verification tests are not automated so there is no regression tests left over
- Quick and Dirty Unit Testing may not take much longer than manual verification tests
- Quick and Dirty Unit Tests are automated and therefore provide protection against future bugs

There is little excuse not to write Quick and Dirty Unit Tests!

Legacy Software Change Algorithm: Details

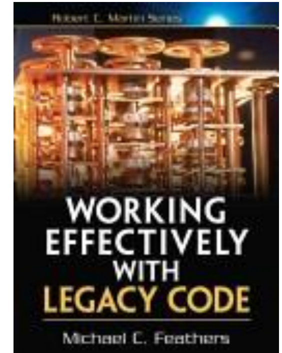
- **Abbreviated Legacy Software Change Algorithm:**
  - 1. Cover code to be changed with tests to protect existing behavior
  - 2. Change code and add new tests to define and protect new behavior
  - 3. Refactor and clean up code to well match current functionality
- **Legacy Code Change Algorithm (Chapter 2 "Working Effectively with Legacy Code")**
  - 1. Identify Change Points
  - 2. Find Test Points
  - 3. Break Dependencies (without unit tests)
  - 4. Cover Legacy Code with (Characterization) Unit Tests
  - 5. Add New Functionality with Test Driven Development (TDD)
  - 6. Refactor to removed duplication, clean up, etc.



- **Covering Existing Code with Tests: Details**
  - Identify Change Points: Find out the code you want to change, or add to
  - Find Test Points: Find out where in the code you can sense variables, or call functions, etc. such that you can detect the behavior of the code you want to change.
  - Break Dependencies: Do minimal refactorings with safer hipper-sensitive editing to allow code to be instantiated and run in a test harness
  - Cover Legacy Code with Unit Tests: If you have the specification for how to code is supposed to work, write tests to that specification. Otherwise, white "Characterization Tests" to see what the code actually does under different input scenarios.

OAK RIDGE National Laboratory

Legacy Software Tools, Tricks, Strategies



- **Reasons to Break Dependencies:**
  - **Sensing:** Sense the behavior of the code that we can't otherwise see
  - **Separation:** Allow the code to be run in a test harness outside of production setting
- **Faking Collaborators:**
  - **Fake Objects:** Impersonates a collaborator to allow sensing and control
  - **Mock Objects:** Extended Fake object that asserts expected behavior
- **Seams:** Ways to inserting test-related code or putting code into a test harness.
  - **Preprocessing Seams:** Preprocessor macros to replace functions, replace header files, etc.
  - **Link Seams:** Replace implementation functions (program or system) to define behavior or sense changes.
  - **Object Seams:** Define interfaces and replace production objects with mock or fake objects in test harness.
  - **NOTE: Prefer Object Seams to Link or Preprocessing Seams!**
- **Unit Test Harness Support:**
  - **C++: Teuchos Unit Testing Tools, Gunit, Boost?**
  - **Python: pyunit ???**
  - **CMake: ???**
  - **Other: Make up your own quick and dirty unit test harness or support tools as needed!**
- **Refactoring and testing strategies … See the book …**

OAK RIDGE National Laboratory

Two Ways to Change Software

The Goal: Refactor five functions on a few interface classes and update all subclass implementations and client calling code. Total change will involve changing about 30 functions on a dozen classes and about 300 lines of client code.

Option A: Change all the code at one time testing only at the end

- Change all the code rebuilding several times and documentation in one sitting **[6 hours]**
- Build and run the tests (which fail) **[10 minutes]**
- Try to debug the code to find and fix the defects **[1.5 days]**
- [Optional] Abandon all of the changes because you can't fix the defects

Option B: Design and execute an incremental and safe refactoring plan

- Design a refactoring plan involving several intermediate steps where functions can be changed one at a time **[1 hour]**
- Execute the refactoring in 30 or so smaller steps, rebuilding and rerunning the tests each refactoring iteration **[15 minutes per average iteration, 7.5 hours total]**
- Perform final simple cleanup, documentation updates, etc. **[2 hour]**

Are these scenarios realistic?

=> This is exactly what happened to me in a Thyra refactoring a few years ago!

Example of Planned Incremental Refactoring

```
// 2010/08/22: rabartl: To properly handle the new SolveCriteria struct with
 // redution functionals (bug 4915) the function solveSupports() must be
 // refactored.  Here is how this refactoring can be done incrementally and
 // safely:
 //
 // (*) Create new override solveSupports(transp, solveCriteria) that calls
 // virtual solveSupportsNewImpl(transp, solveCriteria).
 //
// (*) One by one, refactor existing LOWSB subclasses to implement
 // solveSupportsNewImpl(transp, solveCriteria).  This can be done by
 // basically copying the existing solveSupportsSolveMeasureTypeImpl()
 // override.  Then have each of the existing
 // solveSupportsSolveMeasureTypeImpl() overrides call
 // solveSupportsNewImpl(transp, solveCriteria) to make sure that
 // solveSupportsNewImpl() is getting tested right away.  Also, have the
 // existing solveSupportsImpl(...) overrides call
 // solveSupportsNewImpl(transp, null).  This will make sure that all
 // functionality is now going through solveSupportsNewImpl(...) and is
 // getting tested.
 //
 // (*) Refactor Teko software.
 //
 // (*) Once all LOWSB subclasses implement solveSupportsNewImpl(transp,
 // solveCriteria), finish off the refactoring in one shot:
 //
 //   (-) Remove the function solveSupports(transp), give solveCriteria a
 //   default null in solveSupports(transp, solveCriteria).
 //
 //   (-) Run all tests.
 //
 //   (-) Remove all of the solveSupportsImpl(transp) overrides, rename solve
 //   solveSupportsNewImpl() to solveSupportsImpl(), and make
 //   solveSupportsImpl(...) pure virtual.
...
```

```
 ...
 //
 //   (-) Change solveSupportsSolveMeasureType(transp, solveMeasureType)
to
 //   call solveSupportsImpl(transp, solveCriteria) by setting
 //   solveMeasureType on a temp SolveCriteria object.  Also, deprecate the
 //   function solveSupportsSolveMeasureType(...).
 //
 //   (-) Run all tests.
 //
 //   (-) Remove all of the eixsting solveSupportsSolveMeasureTypeImpl()
 //   overrides.
 //
 //   (-) Run all tests.
 //
 //   (-) Clean up all deprecated working about calling
 //   solveSupportsSolveMeasureType() and instead have them call
 //   solveSupports(...) with a SolveCritera object.
 //
 // (*) Enter an item about this breaking backward compatiblilty for existing
 // subclasses of LOWSB.
```

An in-progress Thyra refactoring started back in August 2010

- Adding functionality for more flexible linear solve convergence criteria needed by Aristos-type Trust-Region optimization methods.
- Refactoring of Belos-related software finished to enabled
- Full refactoring will be finished in time.

OAK RIDGE National Laboratory

Summary of TriBITS Lifecycle Model

- **Motivation:**
  - Allow Exploratory Research to Remain Productive
  - Enable Reproducible Research
  - Improve Overall Development Productivity
  - Improve Production Software Quality
  - Better Communicate Maturity Levels with Customers
- **Self Sustaining Software => The Goal of the Lifecycle Model**
  - Open-source
  - Core domain distillation document
  - Exceptionally well testing
  - Clean structure and code
  - Minimal controlled internal and external dependencies
  - Properties apply recursively to upstream software
  - All properties are preserved under maintenance
- **Lifecycle Phases:**
  - 0: Exploratory (EP) Code
  - 1: Research Stable (RS) Code
  - 2: Production Growth (PG) Code
  - 3: Production Maintenance (PM) Code
- **Grandfathering existing Legacy packages into the lifecycle model:**
  - Apply Legacy Software Change Algorithm => Slowly becomes Self-Sustaining Software over time.
  - Add "Grandfathered" prefix to RS, PG, and PM phases.

Summary of Agile Technical Practices/Skills

- **Unit Testing:** Re-build fast and run fast; localize errors; Well supports continuous integration, TDD, etc.
- **System-Level Testing:** Tests on full system or larger integrated pieces; Slower to build and run; Generally does not well support CI or TDD.
- **Quick and Dirty Unit Tests:** Between unit tests and system tests but easier to write than pure unit tests
- **(Unit or Acceptance) Test Driven Development (TDD):** Write a compiling but failing (unit or system or acceptance) test and verify that it fails; Add/change minimal code until the test passes (keeping all other tests passing)
- **Incremental Structured Refactoring:** Make changes to restructure code without changing behavior (or performance, usually); Separate refactoring changes from changes to change behavior
- **Agile-Emergent Design:** Keep the design simple and obvious for the current set of features (not some imagined set of future features); Continuously refactor code as design changes to match current feature set
- **Legacy Software Change Algorithm**
  - 1. Cover code to be changed with tests to protect existing behavior
  - 2. Change code and add new tests to define and protect new behavior
  - 3. Refactor and clean up code to well match current functionality
- **Safe Incremental Refactoring and Design Change Plan:** Develop a plan; Perform refactoring in many small/safe iterations; final cleanup
- **Legacy Software Tools, Tricks, Strategies**

These are real skills that take time and practice to acquire!

What are the Next Steps?

- Can we adopt the TriBITS Lifecycle Model Phases and self assess to start with? Would that be helpful?
- Should we develop metrics for the different lifecycle phases and start to track them for different phase software?
- Are people willing to commit to using the Legacy Software Change Algorithm to grandfather their software into the TriBITS Lifecycle Model?
- How do we teach developers the core skills of unit testing, test driven development, structured incremental refactoring, Agile-emergent design needed to create well tested clean code to allow for Self-Sustaining Software?
- How do we teach developers how to apply the Legacy Software Change Algorithm?
  - Conduct a reading group for "Working Effectively with Legacy Code"?
  - Look at online webinars/presentations (e.g. ???)?
  - Start by teaching a set of mentors that with then teach other developers? (i.e. this is the Lean approach).

Watch for Trilinos Lifecycle and Technical Practices Survey!