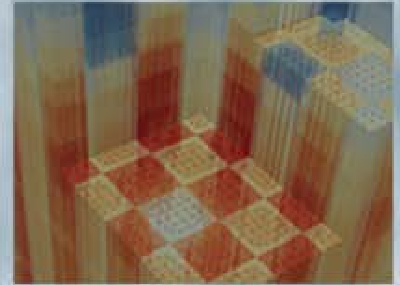
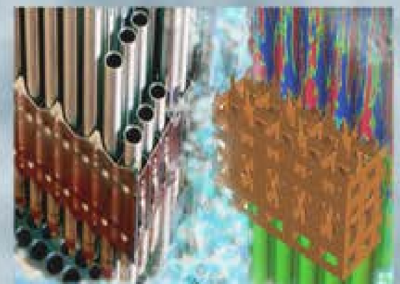
CASL
A DOE Energy Innovation Hub

CASL-I -2015-0102-000



Lean/Agile Principles and Practices for Developing Quality Scientific Software

Roscoe A. Bartlett
Oak Ridge National Laboratory



July 8-10, 2013

U.S. DEPARTMENT OF ENERGY | Nuclear Energy

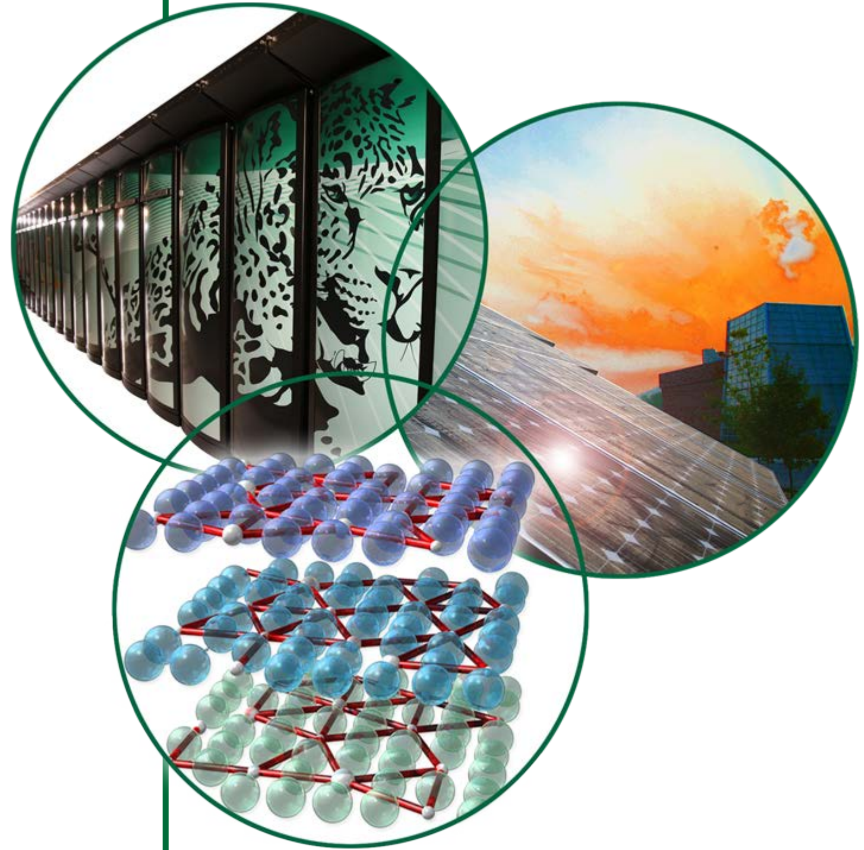
Lean/Agile Principles and Practices for Developing Quality Scientific Software

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

CASL-U-2015-0102-000

OAK RIDGE NATIONAL LABORATORY

MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

The Software Engineering Challenge in Computational Science and Engineering

and in CASL

OAK RIDGE National Laboratory

VERA

VERA GUI/workflow

Reactor Geometry DB

Reactor-Aware FACADEs

VERA Foundation

Data Mapping

Simulation workflow control scripting

Multi-physics coupling toolkit

Simulation post-processing

VUQ (i.e Dakota)

Materials Interface

VERA-Supported Physics Models (e.g. Denovo, Drekar, SCALE, DeCART, ...)

Basic Automatic Mesh Generation

Reactor-aware input

Visualization

Embedded Automatic Mesh Generation

External Physics Models (e.g. StarCCM+, …)

Concrete numerical algorithms (e.g. Epetra, Epetra, ML, Ifpack, …)

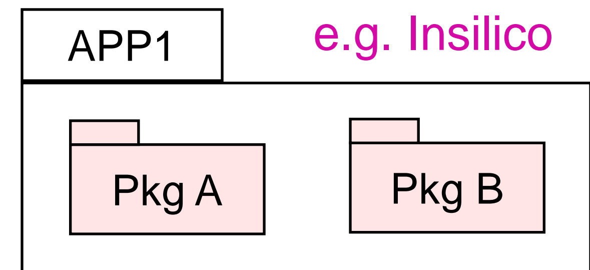
Abstract numerical algorithms (e.g. Thyra, NOX, Rythmos, …)

Advanced Mesh Generation

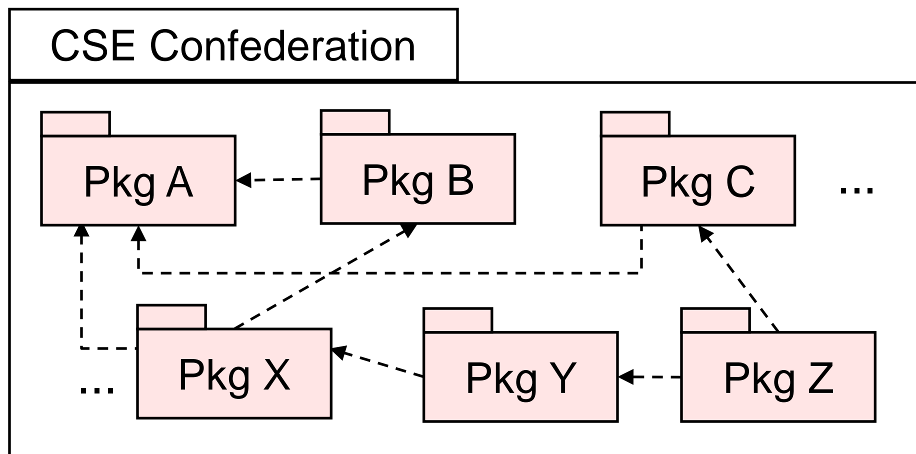
VERA Subdomains and Scope
(NOTE: All relationships are *NOT* shown)

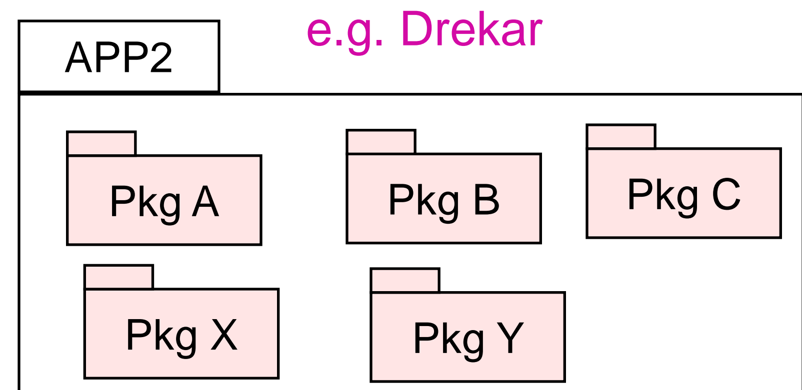
The CSE Software Engineering Challenge

- 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** …
  - ...



e.g. Insilico





e.g. Drekar

CASL is a larger example of this.

Trilinos 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, …)

Key Agile Principles and Technical Practices

Defined: Agile and Lean

- **Agile Software Engineering Methods:**
  - Agile Manifesto (2001) (Capital 'A' in Agile)
  - Founded on long standing wisdom in SE community (40+ years)
  - Push back against heavy plan-driven methods (CMM(I))
  - Focus on incremental design, development, and delivery (i.e. software life-cycle)
  - Close customer focus and interaction and constant feedback
  - Example methods: SCRUM, XP (extreme programming)
  - Becoming a dominate software engineering approach
- **Lean Software Engineering Methods**:
  - Adapted from Lean manufacturing approaches (e.g. the Toyota Production System).
  - Focus on optimizing the value chain, small batch sizes, minimize cycle time, automate repetitive tasks, …
  - Agile methods are seen as a subset of Lean.

References: http:/www.ornl.gov/8vt/readingList.html

Relevant Lean/Agile Principles

- **Agile Design:** Reusable software is best designed and developed by incrementally attempting to reuse it with new clients and incrementally redesigning and refactoring the software as needed keeping it simple.
  - Technical debt in the code is managed through continuous incremental (re)design and refactoring.
- **Agile Quality**: High quality defect-free software is most effectively developed by not putting defects into the software in the first place.
  - High quality software is best developed collaboratively (e.g. pair programming and code reviews).
  - Software is fully verified before it is even written (i.e. Test Driven Development (TDD) for system verification and unit tests).
  - High quality software is developed in small increments and with sufficient testing in between sets of changes.
- **Agile Integration**: Software needs to be integrated early and often
- **Agile Delivery**: Software should be delivered to real (or as real as we can make them) customers is short (fixed) intervals.

References: http:/www.ornl.gov/8vt/readingList.html

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.
- (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)
- Continuous Integration
  - Integrating software in small batches of changes frequently
  - Most development on primary 'master' branch
- 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

Course-Grained 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".

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

Course-Grained "Unit Tests" vs. Manual Tests

Course-Grained Unit Tests

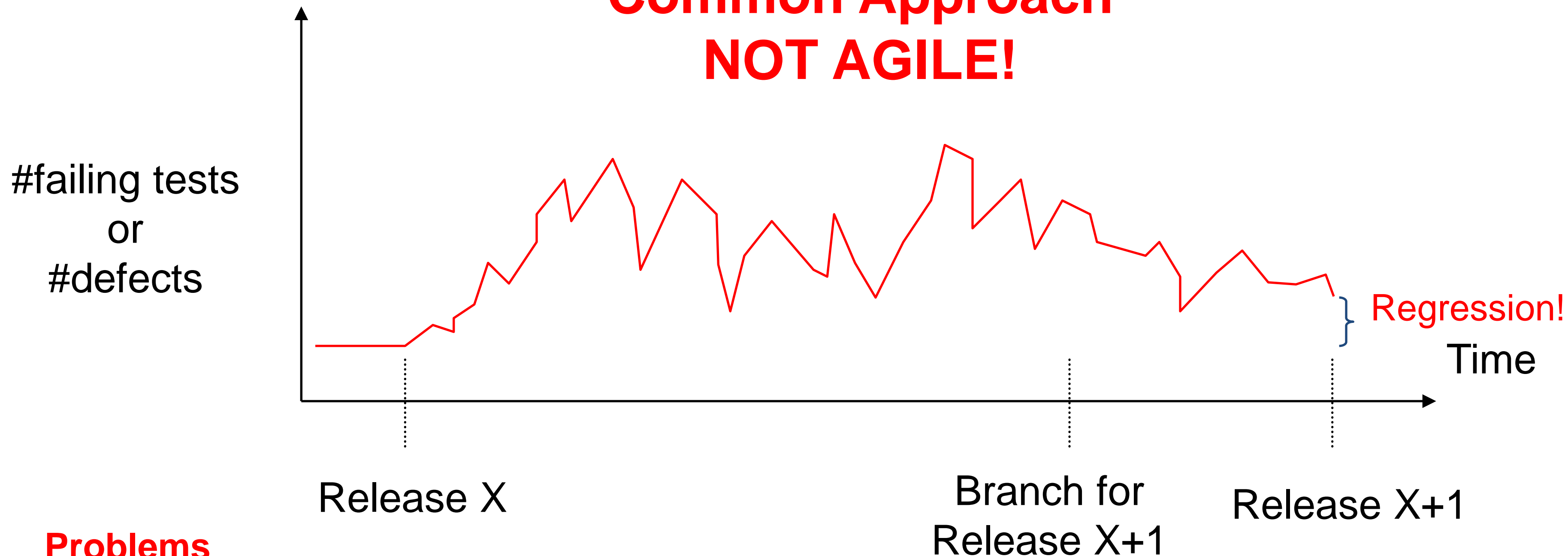
- Courser-grained tests that instantiate several objects and/or execute several functions before checking final results.
- Courser-grained than "unit tests" but much finer grained than "system tests"
- May be written much more quickly than pure "unit tests" but still cover almost the same behavior.
- 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

Course-Grained 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
- Course-Grained Unit Testing may not take much longer than manual verification tests
- Course-Grained Unit Tests are automated and therefore provide protection against future bugs

There is little excuse not to write Course-Grained Unit Tests!

Common Approach: Development Instability

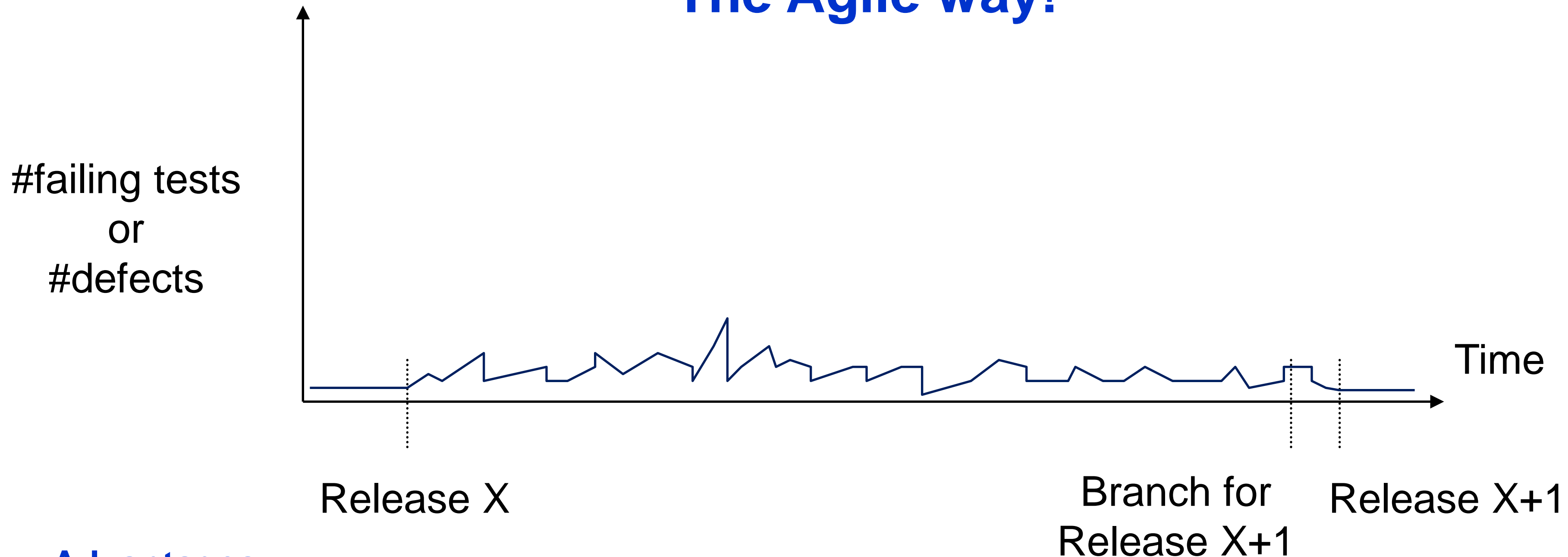


Problems

- 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
- Nearly impossible to consider more frequent development integration models
- High risk of creating a regression

OAK RIDGE National Laboratory

Lean/Agile Approach: Development Stability



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 (Fundamental Principle of Software Quality)
- Allows many options in how to do development integration models

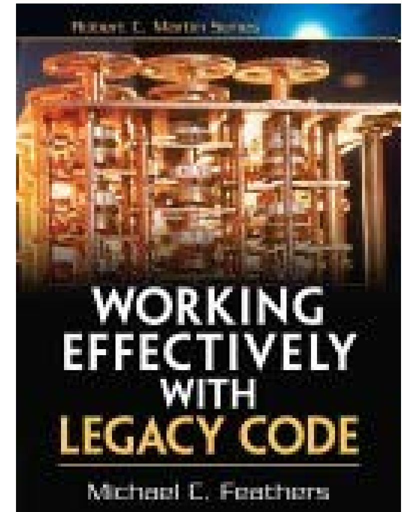
The Legacy Software Change Algorithm

Definition of Legacy Code and Changes

Legacy Code = Code Without Tests

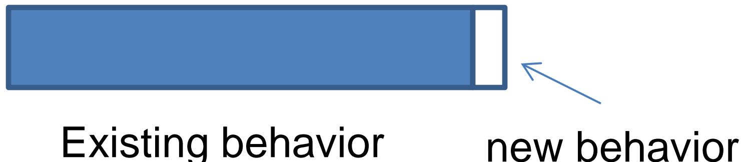
"Code without tests is bad code. It does not matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."

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



Existing behavior new behavior

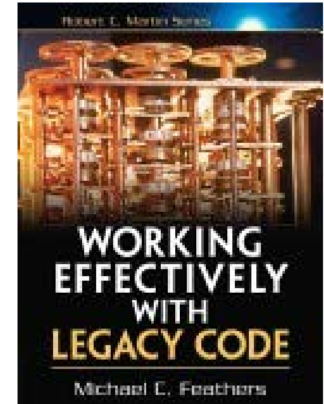
Preserving behavior under change:

"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"

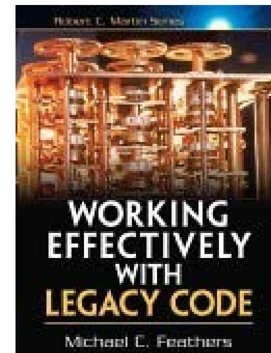
Legacy Software Change Algorithm: Details

- **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.

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 …

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 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.
- **Course-Grained 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!

THE END

OAK RIDGE
National Laboratory