Some Agile Best Technical Practices for the Development of Research-Based CSE Software

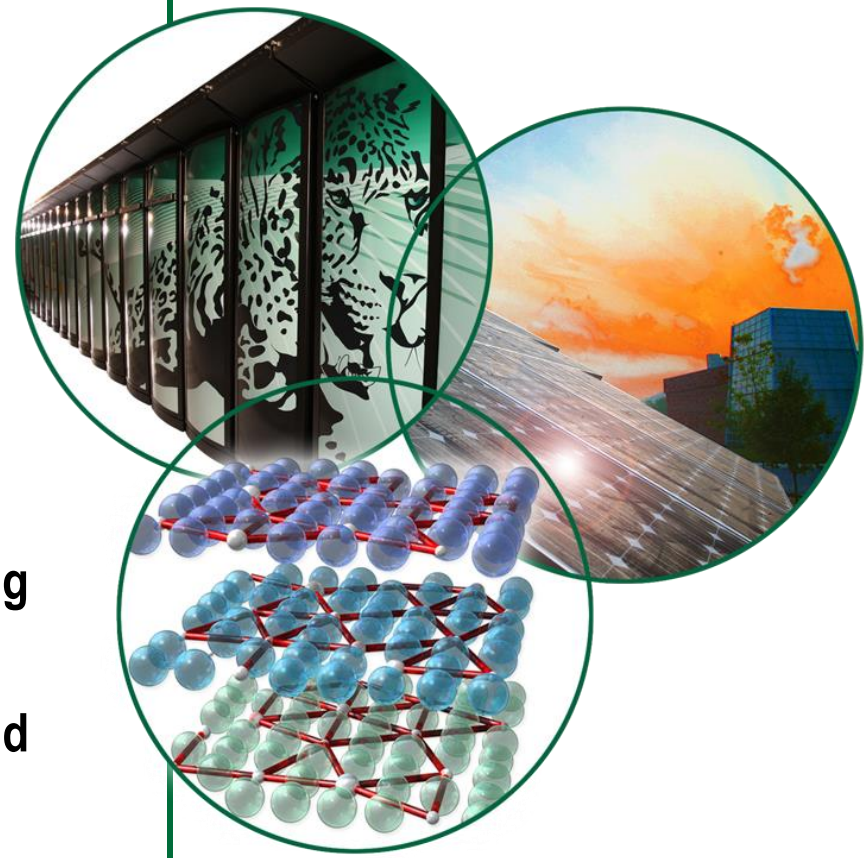
Roscoe A. Bartlett

ORNL Computer Science and Mathematics Div

CASL Physics Integration Software Engineering Lead

Trilinos Software Engineering Technologies and Integration Lead

Website: http://www.ornl.gov/~8vt/



U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY

MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Overview of CASL

- **CASL: C**onsortium for the **A**dvanced **S**imulation of **L**ightwater reactors
- DOE Innovation Hub including DOE labs, universities, and industry partners
- Goals:
  - Advance modeling and simulation of lightwater nuclear reactors
  - Produce a set of simulation tools to model lightwater nuclear reactor cores to provide to the nuclear industry: **VERA: Virtual Environment for Reactor Applications**.
- Phase 1: July 2010 – June 2015
- Phase 2: July 2015 – June 2020
- Organization and management:
  - ORNL is the hub of the Hub
  - Milestone driven (6 month plan-of-records (PoRs))
  - Focus areas: **Physics Integration (PHI)**, Thermal Hydraulic Methods (THM), Radiation Transport Methods (RTM), Advanced Modeling Applications (AMA), Materials Performance and Optimization (MPO), Validation and Uncertainty Quantification (VUQ)

Interoperable Design of Extreme-scale Application Software (IDEAS)

Motivation

Enable *increased scientific productivity*, realizing the potential of extreme-scale computing, through *a new interdisciplinary and agile approach to the scientific software ecosystem.*

Objectives

Address confluence of trends in hardware and increasing demands for predictive multiscale, multiphysics simulations.

Respond to trend of continuous refactoring with efficient agile software engineering methodologies and improved software design.

IDEAS productivity

Desired Outcomes of Interest

- Improved interoperability between Trilinos, PETSc, HYPRE, and SuperLU
- Long-term sustained compatbility between Trilnios, PETSc, HYPRE, and SuperLU
- Development of SE materials targeted to CSE community

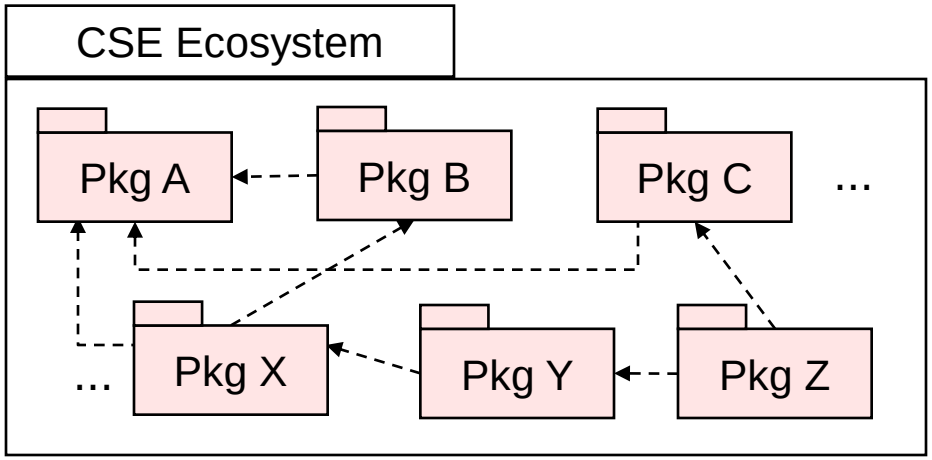
Goals / Objectives

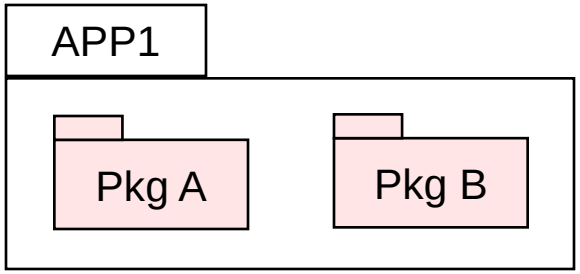
- Improve SW productivity for target BER applications
- Leadership and broader impact to DOE

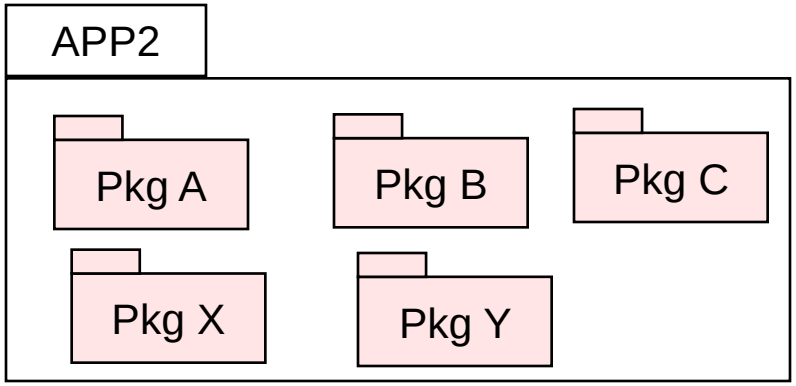
Website: https://ideas-productivity.org/
Resources: https://ideas-productivity.org/resources/
HowTos: https://ideas-productivity.org/resources/howtos/

The CSE Software Ecosystem Challenge

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







Trilinos is a smaller example

CASL and IDEAS are larger examples

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, …)

Software Practices and Research Software?

Even research software (only written by and run by a researcher) needs to have a base level of SE practices/quality to support basic research!

Example: A simple software defect in protein configuration/folding research code [1]

- The researcher:
  - A respected Presidential Early Career winner
- The defect:
  - An array index failure (i.e. verification failure) leading to incorrect protein folding (validation failure)
- Direct impact:
  - Several papers were published with incorrect
- Indirect impact:
  - Papers from other competing researchers with different results were rejected
  - Proposals from other researchers with different results were turned down
- Final outcome: Defect was finally found and author retracted five papers!
  - But: Damage to the research community not completely erased!

And Software Development Productivity Matters CSE Researchers Too!

[1] Miller, G. “A Scientist's Worst Nightmare: Software Problem Leads to Five Retractions”, Science, vol 314, number 5807, Dec. 2006, pages 1856-1857

Key Agile Principles and Technical Practices

Defined: Agile

- **Agile Software Engineering Methods:**
  - Agile Manifesto (2001) (Capital ‘A’ in Agile)
  - Founded on long standing wisdom in SE community (45+ 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)
  - Has a dominate software engineering approach (example IBM)

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

Principles for Agile Technical Practices

- **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)).
  - 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

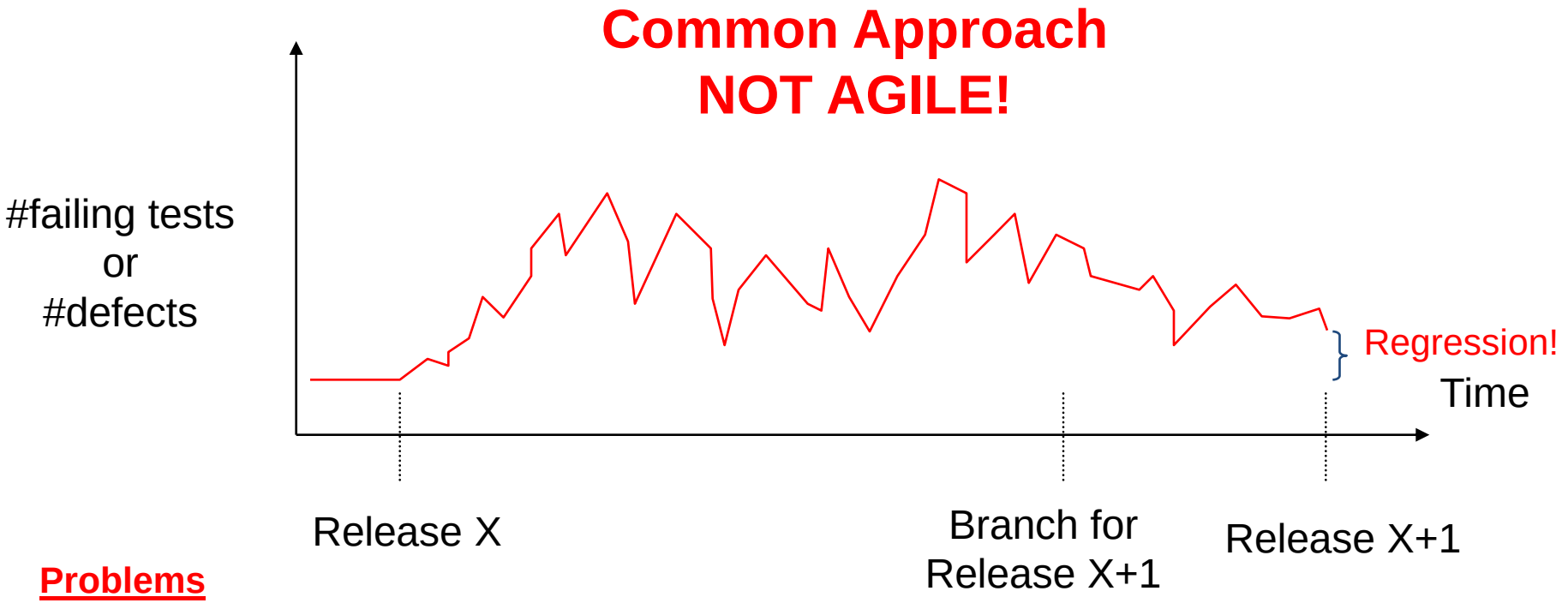
- **Unit Testing**
  - Re-build fast and run fast
  - Localize errors
  - Well supports continuous integration, TDD, etc.
- **Integration and System-Level Testing**
  - Tests on full system or larger integrated pieces
  - Slower to build and run
  - Generally does not well support CI or TDD.
- **Test Driven Development (TDD)**
  - Write a compiling but failing 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

Integration Tests

Courser-grained "Integration tests" can be 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!

Common Approach: Development Instability

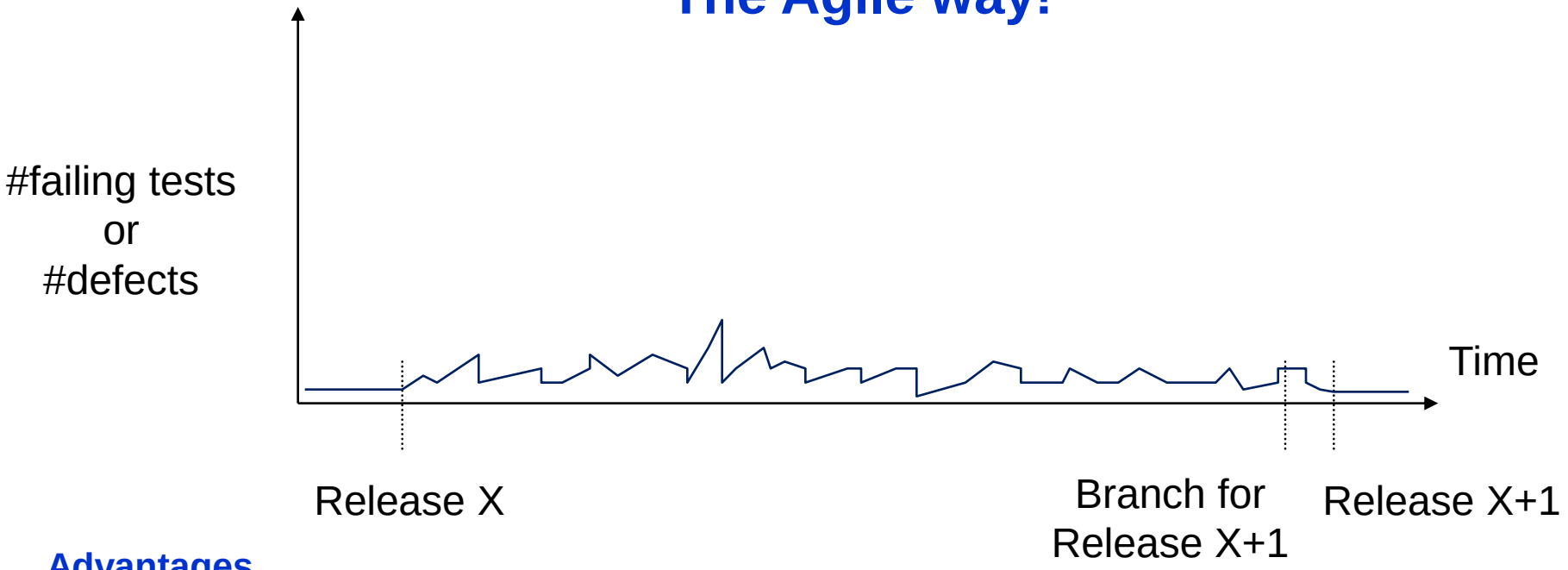


Problems

- 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

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

Software Testing

Definition and Categorization of Tests

Granularity of Tests

- **Unit**: Single function or small class/object. Builds fast, runs fast, isolates errors
  => 100% coverage easy, makes refactoring and debugging easy but can take more time to write and maintain
- **Integration**: Several functions on several objects, etc.. Can build/run pretty fast but may not isolate errors well
  => Harder to debug than unit tests but easier than system tests. But easier to write and maintain than unit tests
- **System**: Runs full system seen by user. Can take long time to build and run, errors may be in 100K lines of code
  => Lower coverage, makes refactoring and debugging very hard but tests user-oriented use cases

Types of Tests (i.e. what is being tested)

- **Verification:** Tests against an internal requirement of the code. Examples: Number of iterations of CG == number of unique eigen values. Monotonic decrease in residual for GMRES.
  => Usually robust / does not break if code is improved but can be harder to write
- **Acceptance:** Tests against external user requirements. Formal UQ-based validation an example.
- **No-change** (characterization): Just tests that the output of the code does not change. Often (incorrectly) called "regression" tests.
  => Tend to be fragile and break even when the code is improved but easiest types of test to write
- **Performance**: Tests usage of resources (memory, CPU time, etc.). Example: Catch if new version is slower.
  => Catches problems missed by "correctness" tests but can be hard to write, maintain, and flag real regressions

Other Definitions

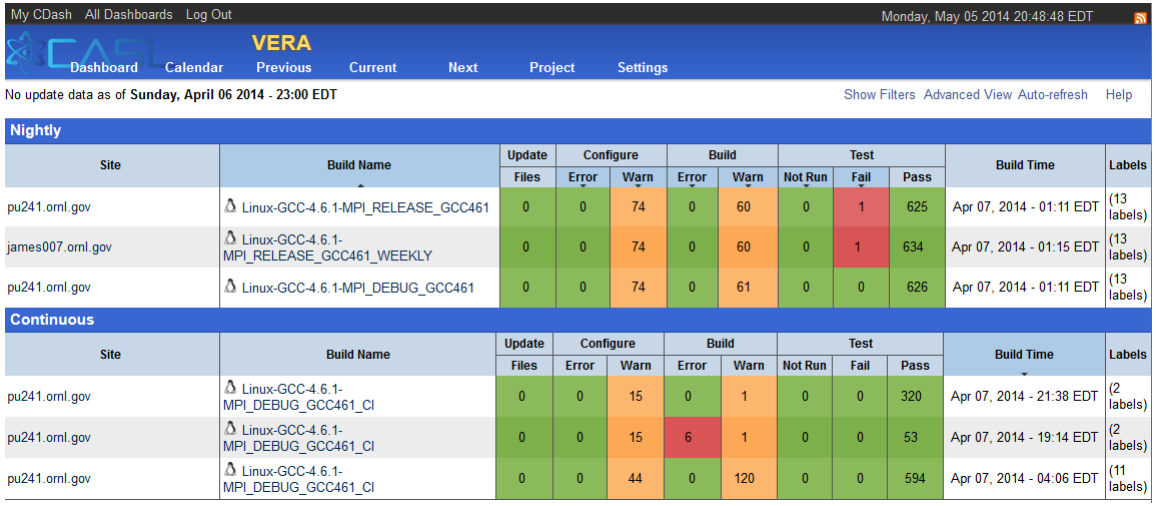
- **Regression test suite**: Set of tests (verification, acceptance, no-change, performance, …) run to check if code is still performing as it did in prior versions.

See 4-page document "Definition and Categorization of Tests for CSE Software"
https://ideas-productivity.org/resources/howtos/

Testing Support Tools

Test Harness Frameworks

- **System-level test harness**: Run system executable test that report results appropriately
  - Display results dashboards
  - Send out emails for failures
  - Examples: CTest/CDash, Jenkins
- **Unit test harness:** Effectively define and run finer-grained integration and unit tests as single programs. Examples: Google Test, pFUnit, Trilinos/Teuchos Unit Test Harness



Selecting good tools and adapting them to your project/environment can massively improve productivity!

Testing Analysis Tools (Run on an existing test suite)

- **Code Coverage:** Produce reports of code covered/uncovered (by line, branch, function, file, directory, package, etc.). Examples: lcov/gcov, bulleye
- **Memory usage error detection:** Detects array bounds errors, pointer errors, memory leaks, etc. Examples: valgrind, purify, Clang tools.

See 2-page document "What Are Software Testing Practices?"
https://ideas-productivity.org/resources/howtos/

How To Add Tests/Features to Existing Code

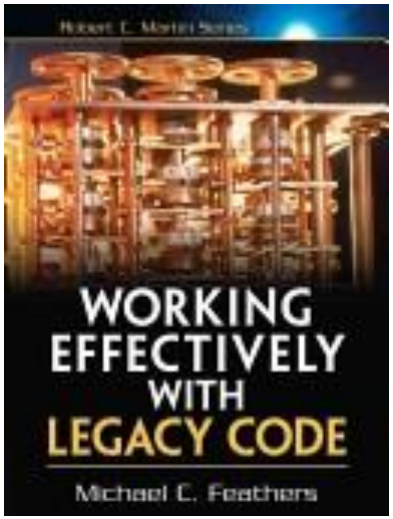
1. **Set up automated builds** of the code with **high warning levels** and **eliminate all warnings**
2. **Select system-level test harness** (e.g. CTest/CDash, Jenkins) and **unit test harnesses** (e.g. Google Test, pfUnit) and **adapt to your environment**
3. **Add system-level tests to protect major user functionality** (no-change and verification tests if possible)
4. **Add integration and unit tests** (as needed for adding/changing code)
    1. **Incorporate tests to protect code to be changed** (see Legacy Software Change Algorithm)
    2. **Add new features or fix bugs with tests** (test-driven development (**TDD**))
        1. Add new tests that define desired behavior (feature or bug).
        2. Run new tests and verify they fail (**RED**).
        3. Add the minimal code to get new tests to pass (**GREED**).
        4. Refactor the covered code to clean up and remove duplication (**REFACTOR**).
        5. Review all changes to existing code, new code and new tests.
5. **Select code coverage** (e.g., gcov/lcov) and **memory usage error detection** (e.g., valgrind) analysis tools.
6. **Define regression test suites**
    1. Define a faster-running **pre-push regression test suite** (e.g., single build with faster running tests) and run it before every push.
    2. Define a more comprehensive **nightly regression test suite** (e.g., builds and all tests on several platforms and compilers, code coverage, and memory usage error detection) and run every night.
7. Have a **policy of 100% passing pre-push regression tests** and work hard to maintain that.
8. Work to **fix all failing nighty regression tests on a reasonable schedule**

See 2-page document "How to Add and Improve Testing in Your CSE Software Project"
https://ideas-productivity.org/resources/howtos/

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

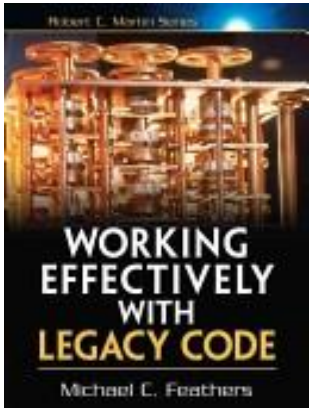
Preserving behavior under change:

"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"

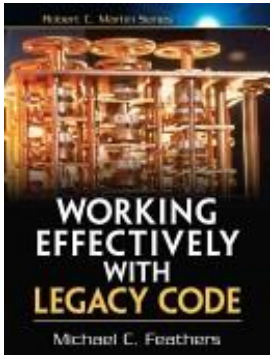
Legacy Software Change Algorithm: Details

- **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 Code with Verification or No-change/Characterization Unit or Integration Tests
  - 5. Add New Functionality with Test Driven Development (TDD)
  - 6. Refactor to remove 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. Can be at unit or integration test levels (consider tradeoffs).
  - **Cover Legacy Code with Unit Tests**: If you have the specification for how to code is supposed to work, write tests to that specification (i.e. verification tests). Otherwise, write no-change or "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 …**

Incremental Structured Refactoring

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.

Lean/Agile, Lifecycle Models, TriBITS

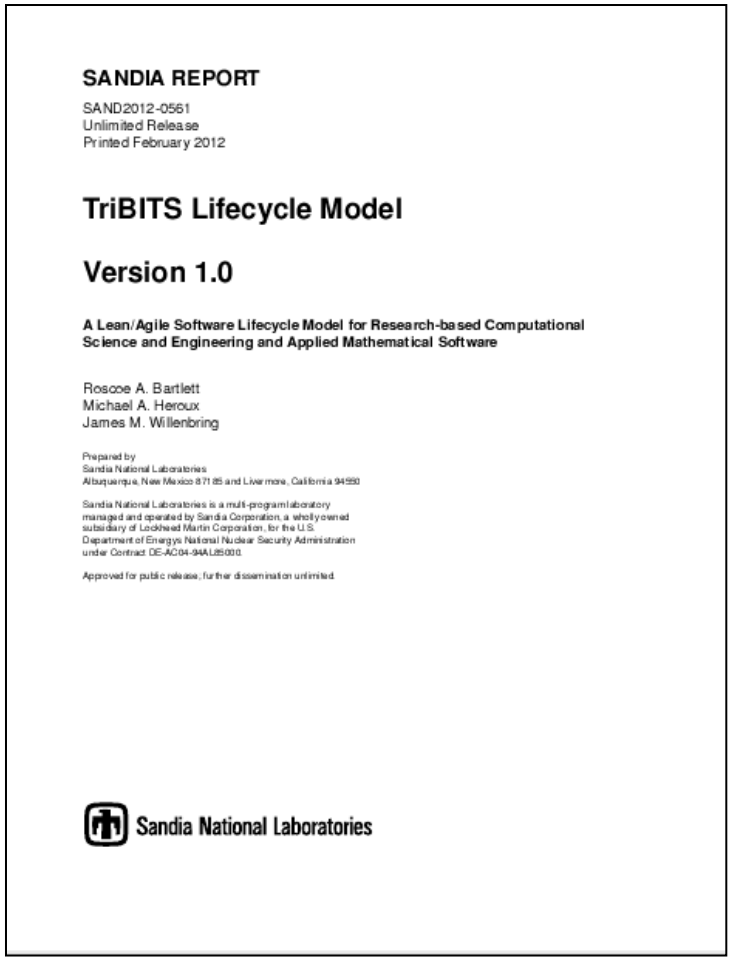
Defined: Life-Cycle, Agile and Lean

- **Software Life-Cycle:** The processes and practices used to design, develop, deliver and ultimately discontinue a software product or suite of software products.
  - Example life-cycle models: Waterfall, Spiral, Evolutionally Prototype, Agile, …
- **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

Overview 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)**
  - 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.



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.

Sandia National Laboratories

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:

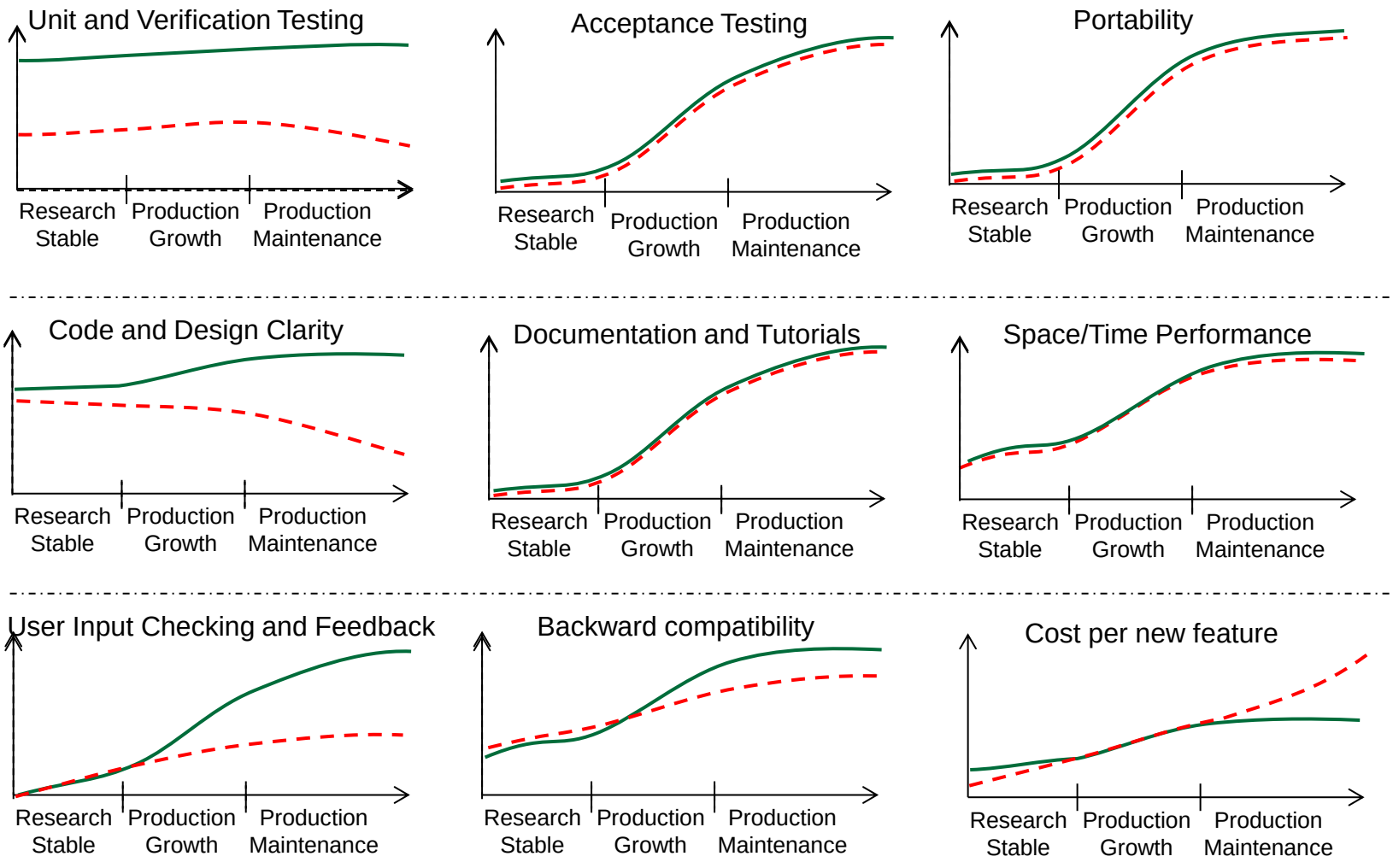
- Assuming customer validation of code is easy (i.e. linear or nonlinear algebraic equation solvers => compute the residual) …
- Can be very fast to initially create new code
- Works for the customers code right away

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 or 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!

TriBITS Lifecycle Model (–) vs. VCA (--)



Software Engineering and HPC
Efficiency vs. Other Quality Metrics

How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑		↑	↑			↑	↓
Usability		↑				↑	↑	
Efficiency	↓		↑	↓	↓	↓	↓	
Reliability	↑			↑	↑		↑	↓
Integrity			↓	↑	↑			
Adaptability					↓	↑		↑
Accuracy	↑		↓	↑		↓	↑	↓
Robustness	↓	↑	↓	↓	↓	↑	↓	↑

Source:
Code Complete 2nd Edition
Steve McConnell

Correctness == Verification
Accuracy == Validation

⇓

Efficiency improvement efforts can make Verification & Validation harder!

Helps it ↑

Hurts it ↓

Summary of Agile Technical Practices/Skills

- **Unit Testing:** Re-build fast and run fast; localize errors; Well supports continuous integration, TDD, etc.
- **Integration-level Testing:** Test of several objects/functions together; between unit and system-level
- **System-Level Testing:** Tests on full system or larger integrated pieces; Slower to build and run; Generally does not well support CI or TDD.
- **Test Driven Development (TDD):** Write a compiling but failing 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