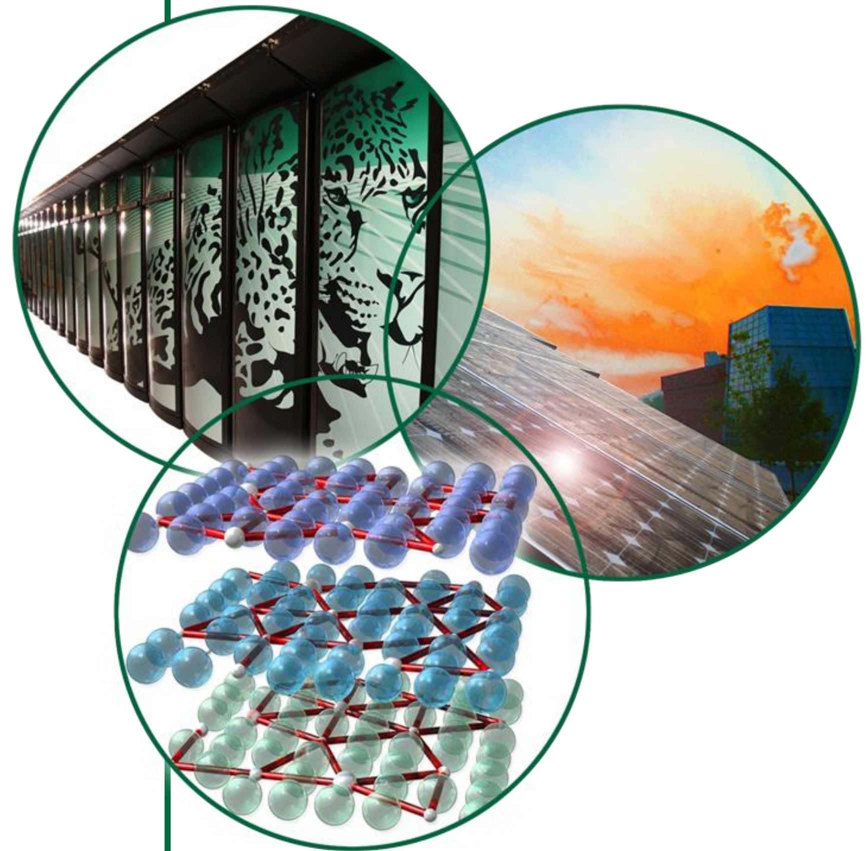
TriBITS Lifecycle Model and Agile Technical Practices for Trilinos?

Trilinos Spring Developers Meeting

May 22, 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

TriBITS Lifecycle Model Overview

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

0: Exploratory (EP) Code

1: Research Stable (RS) Code

2: Production Growth (PG) Code

3: Production Maintenance (PM) Code

-1: Unspecified Maturity (UM) Code

0: Exploratory (EP) Code

- Primary purpose is to explore alternative approaches and prototypes, not to create software.
- Generally not developed in a Lean/Agile consistent way.
- Does not provide sufficient unit (or otherwise) testing to demonstrate correctness.
- Often has a messy design and code base.
- Should not have customers, not even "friendly" customers.
- No one should use such code for anything important (not even for research results, but in the current CSE environment the publication of results using such software would ikely still be allowed).
- Generally should not go out in open releases (but could go out in releases and is allowed by this lifecycle model).
- Does not provide a direct foundation for creating production-quality code and should be put to the side or thrown away when starting product development

1: Research Stable (RS) Code

- Developed from the very beginning in a Lean/Agile consistent manner.
- Strong unit and verification tests (i.e. proof of correctness) are written as the code/algorithms are being developed (near 100% line coverage).
- Has a very clean design and code base maintained through Agile practices of emergent design and constant refactoring.
- Generally does not have higher-quality documentation, user input checking and feedback, space/time performance, portability, or acceptance testing.
- Would tend to provide for some regulated backward compatibility but might not.
- Is appropriate to be used only by "expert" users.
- Is appropriate to be used only in "friendly" customer codes.
- Generally should not go out in open releases (but could go out in releases and is allowed by this lifecycle model).
- Provides a strong foundation for creating production-quality software and should be the first phase for software that will likely become a product.

2: Production Growth (PG) Code

- Includes all the good qualities of Research Stable code.
- Provides increasingly improved checking of user input errors and better error reporting.
- Has increasingly better formal documentation (Doxygen, technical reports, etc.) as well as better examples and tutorial materials.
- Maintains clean structure through constant refactoring of the code and user interfaces to make more consistent and easier to maintain.
- Maintains increasingly better regulated backward compatibility with fewer incompatible changes with new releases.
- Has increasingly better portability and space/time performance characteristics.
- Has expanding usage in more customer codes.

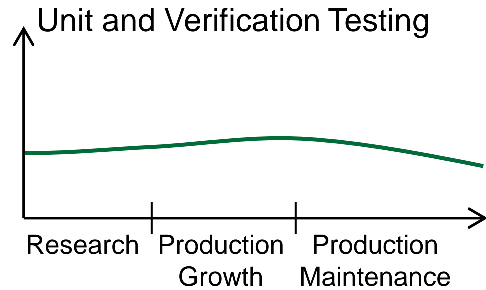
3: Production Maintenance (PM) Code

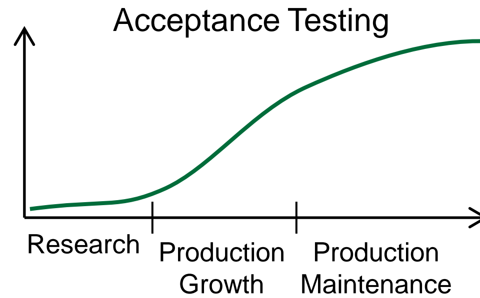
- Includes all the good qualities of Production Growth code.
- Primary development includes mostly just bug fixes and performance tweaks.
- Maintains rigorous backward compatibility with typically no deprecated features or any breaks in backward compatibility.
- Could be maintained by parts of the user community if necessary (i.e. as “self-sustaining software”).

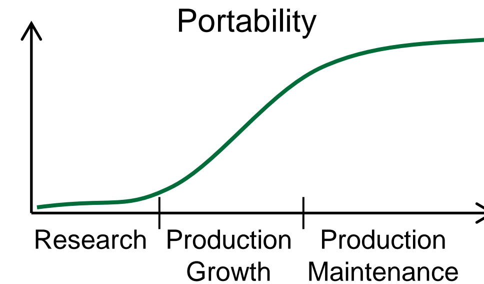
-1: Unspecified Maturity (UM) Code

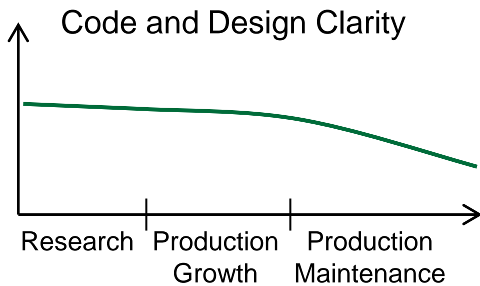
- Provides no official indication of maturity or quality
- i.e. “Opt Out” of the TriBITS Lifecycle Model

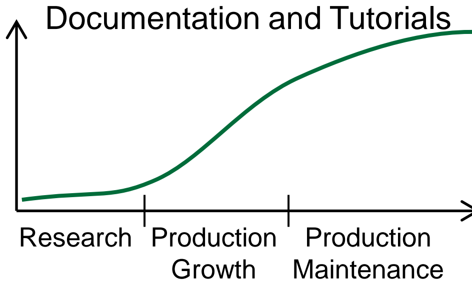
Typical (i.e. non-Lean/Agile) CSE Lifecycle

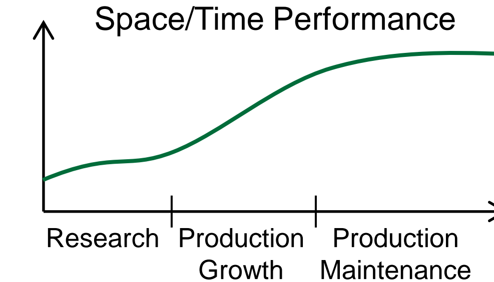


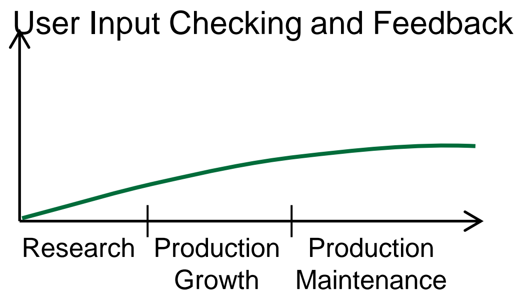


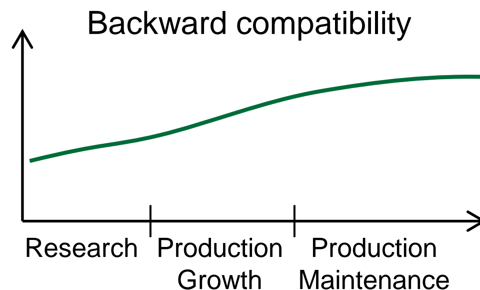


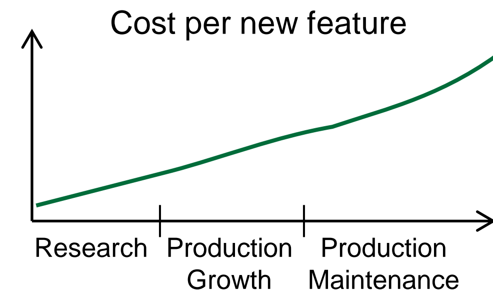




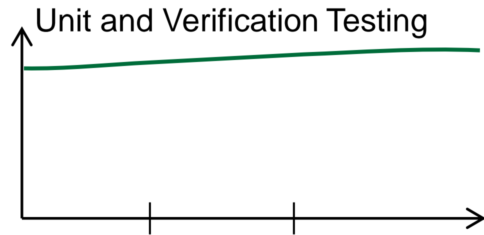


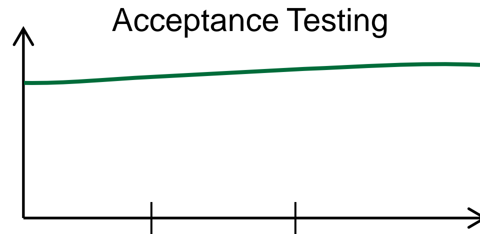


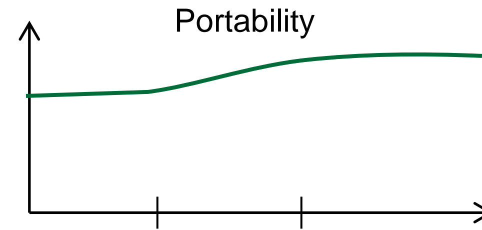


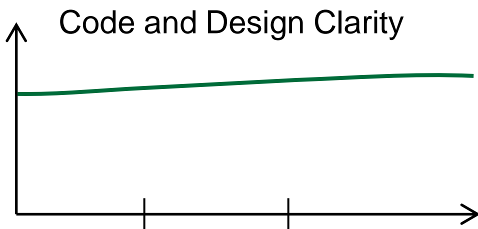


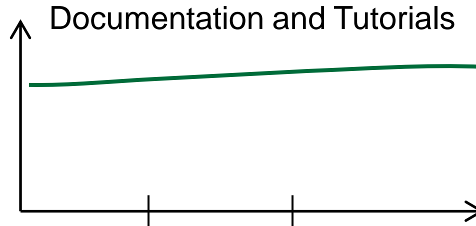
Pure Lean/Agile Lifecycle: "Done Done"

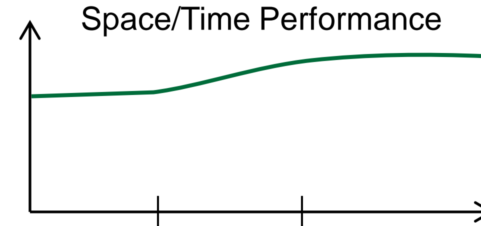


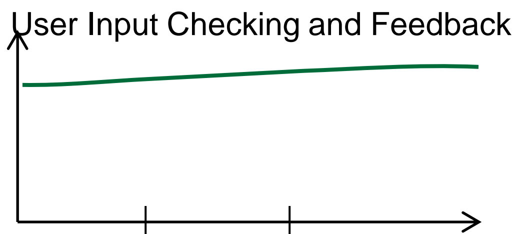


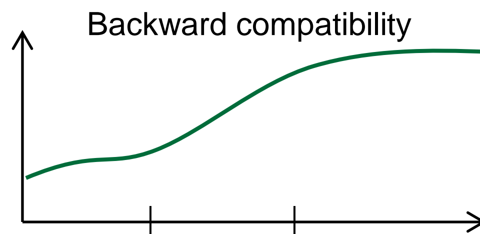


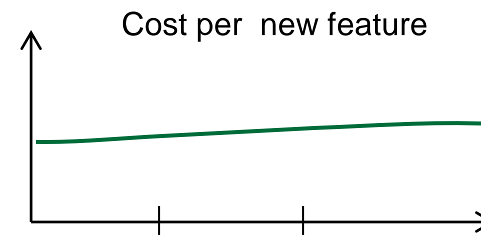




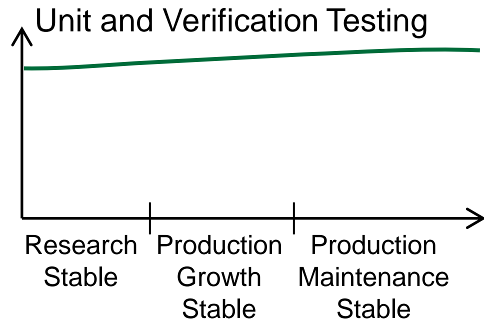


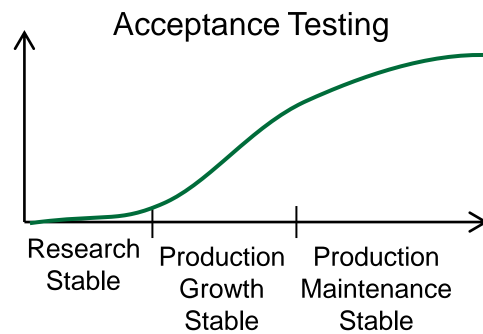


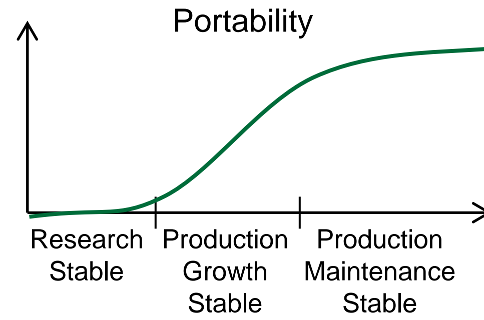


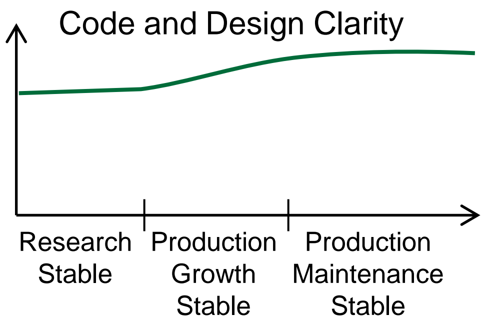


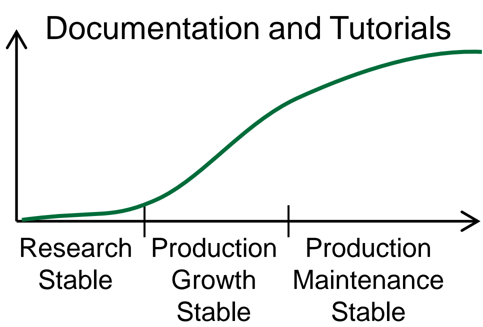
Proposed TriBITS Lean/Agile Lifecycle

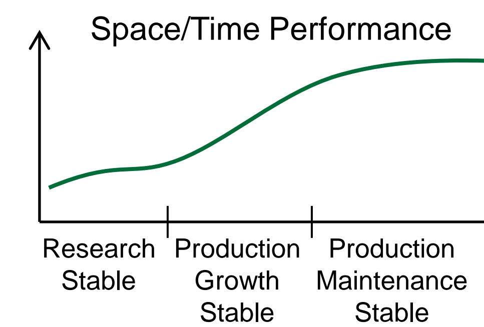


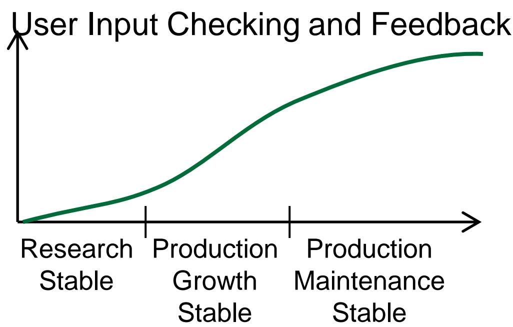


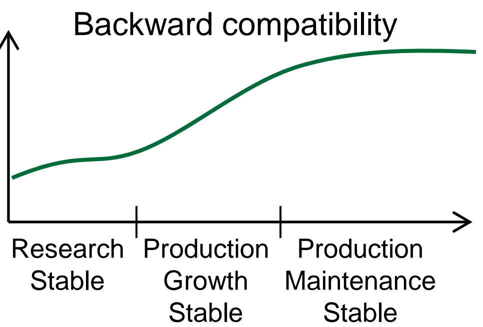


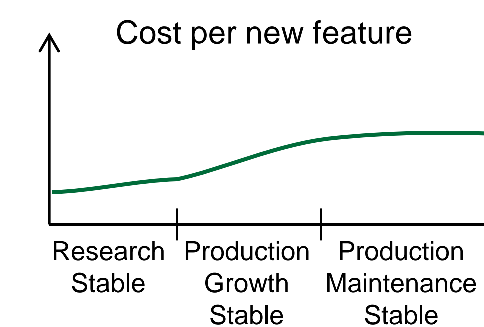












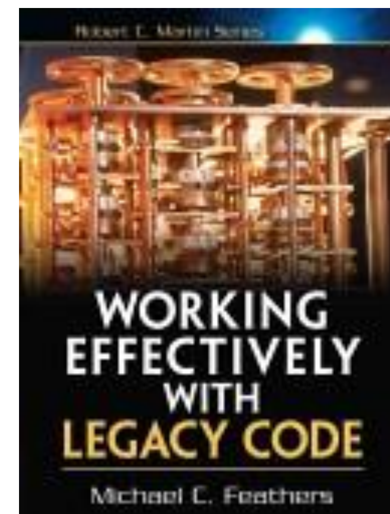
End of Life?

Long-term maintenance and end of life issues for Self-Sustaining Software:

- **User community can help to maintain it**
- **If the original development team is disbanded, users can take parts they are using and maintain it long term**
- **Can stop being built and tested if not being currently used**
- **However, if needed again, software can be resurrected, and continue to be maintained**

NOTE: Distributed version control using tools like Git and Mercurial greatly help in reducing risk and sustaining long lifetime.

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

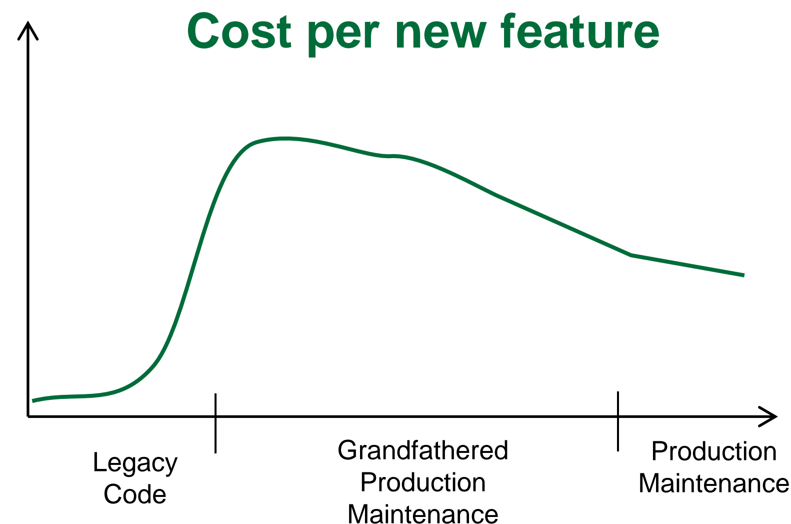
3. Refactor and clean up code to well match current functionality

Grandfathered Lifecycle Phases:

1. Grandfathered Research Stable (GRS) Code

2. Grandfathered Production Growth (GPG) Code

3. Grandfathered Production Maintenance (GPM) Code



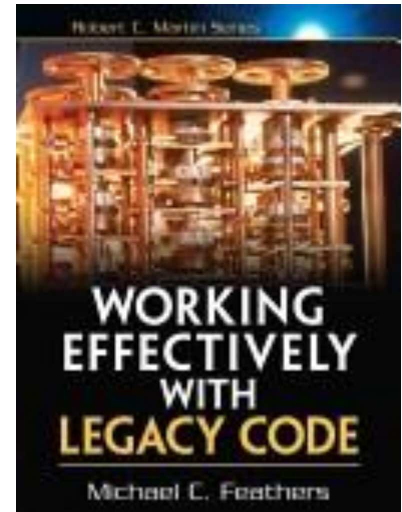
NOTE: After enough iterations of the Legacy Software Change Algorithm the software may approach Self-Sustaining software and be able to remove the "Grandfathered" prefix!

Key Agile Technical Concepts, Practices and Skills

OAK RIDGE
National Laboratory

Definition of Legacy Code and Changes

Legacy Code = Code Without Tests

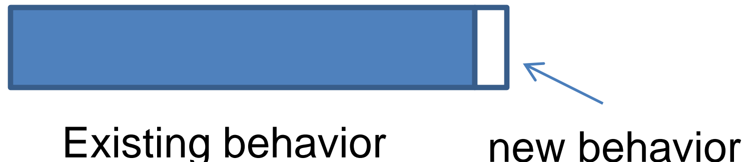


"Code without tests is bad code. It does not matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our ode is getting better or worse."

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

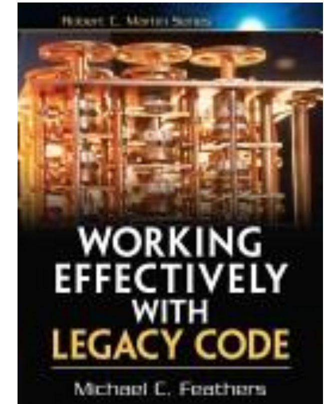
Quick and Dirty Unit Testing vs. Manual Verification Tests

- 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!

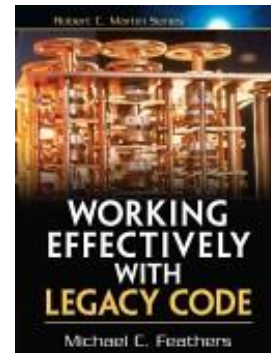
Legacy Software Change Algorithm: Details

- **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.

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.

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!

THE END