TriBITS Lifecycle Model : Version 1.0

Dr. Roscoe A. Bartlett, Ph.D.

CASL Vertical Reactor Integration Software Engineering Lead

Trilinos Software Engineering Technologies and Integration Lead

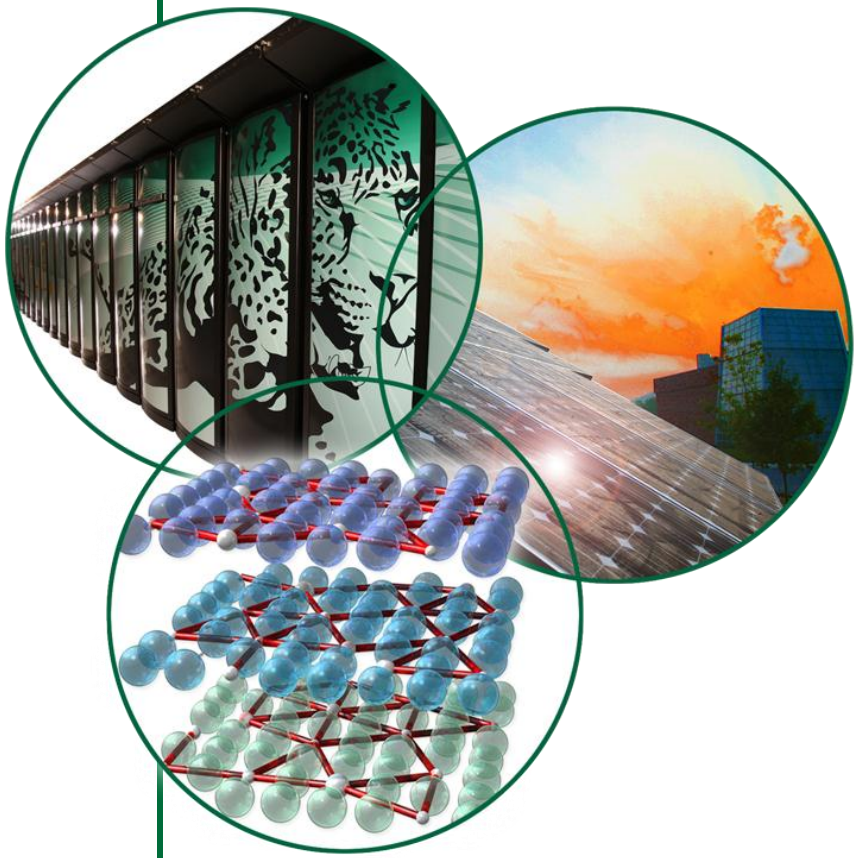
Computational Eng. & Energy Sciences

Computer Science and Mathematics Div

Co-authors:

Mike Heroux (SNL)

Jim Willenbring (SNL)

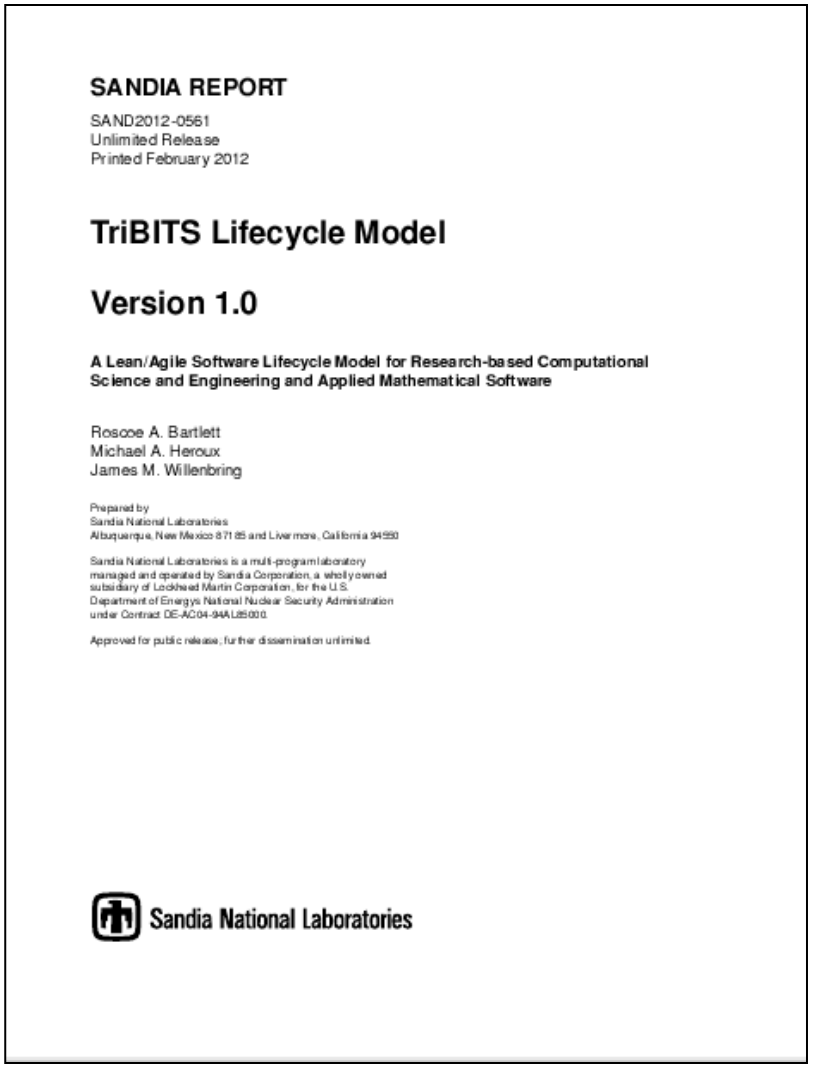


U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY

MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

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

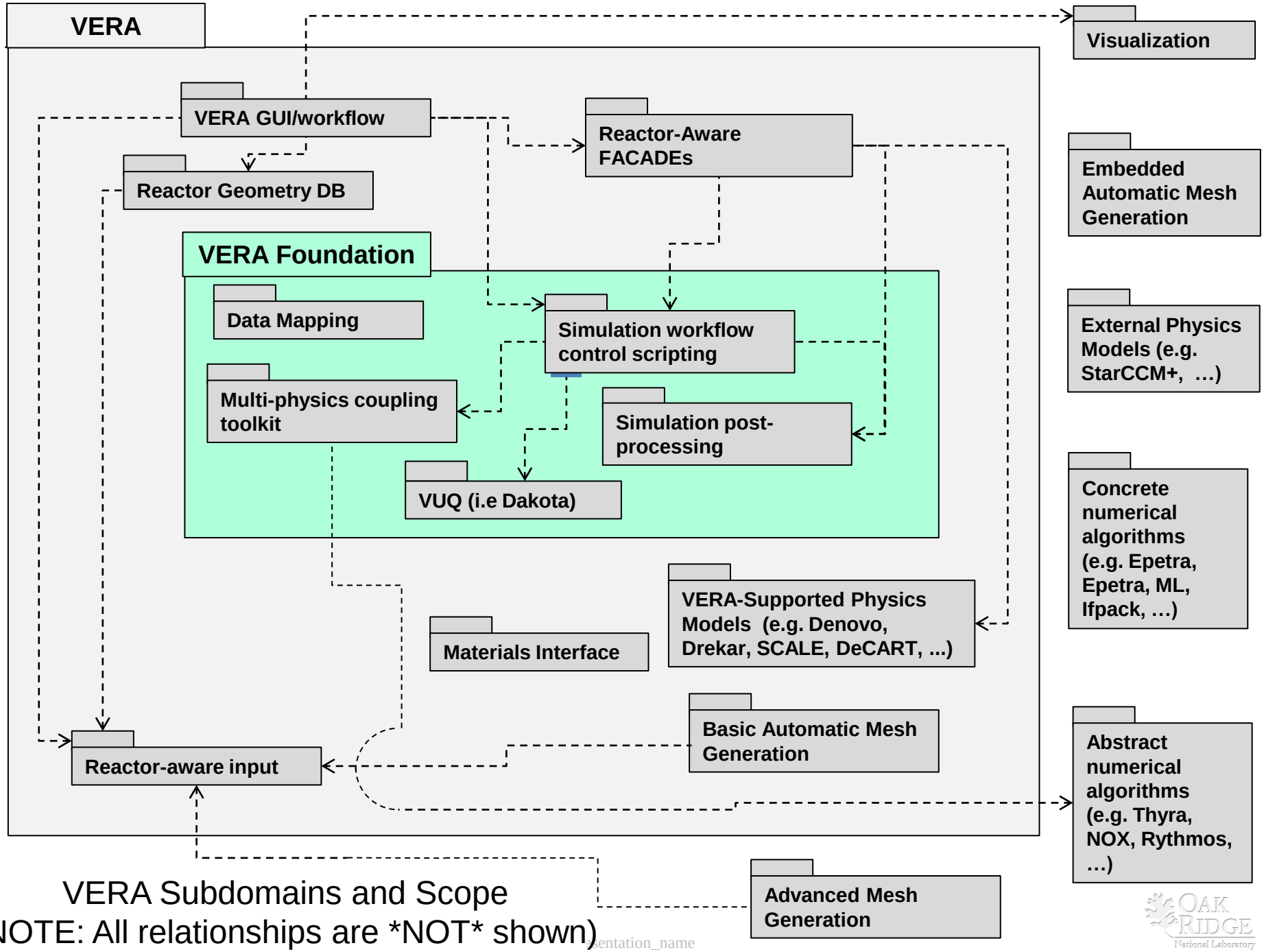
Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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.

Sandia National Laboratories

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

Motivation for the TriBITS Lifecycle Model



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

Overview of Trilinos

- Provides a suite of numerical solvers and discretization methods so support predictive simulation.
- Provides a decoupled and scalable development environment to allow for algorithmic research and production capabilities => "Packages"
- Mostly C++ with some C, Fortran, Python …
- Advanced object-oriented and generic C++ …
- Freely available under open-source BSD and LGPL licenses ...

Current Status

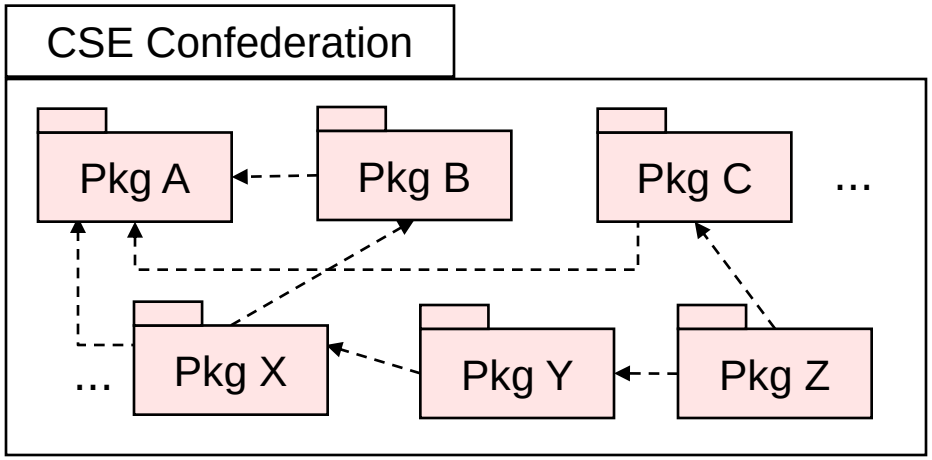
- Current Release Trilinos 10.12 (July 2012)
- Next Release Trilinos 11.0 (September 2012)

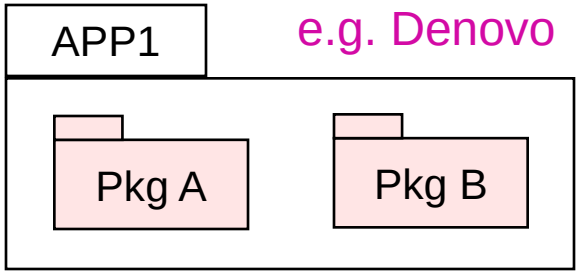
Trilinos website

http://trilinos.sandia.gov

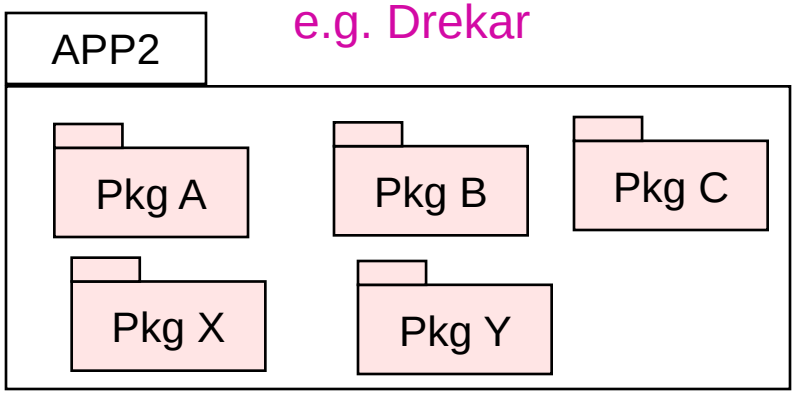
The CSE Software Engineering Challenge

- 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. Denovo



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++)

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, …)

Background

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 fall under 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

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:

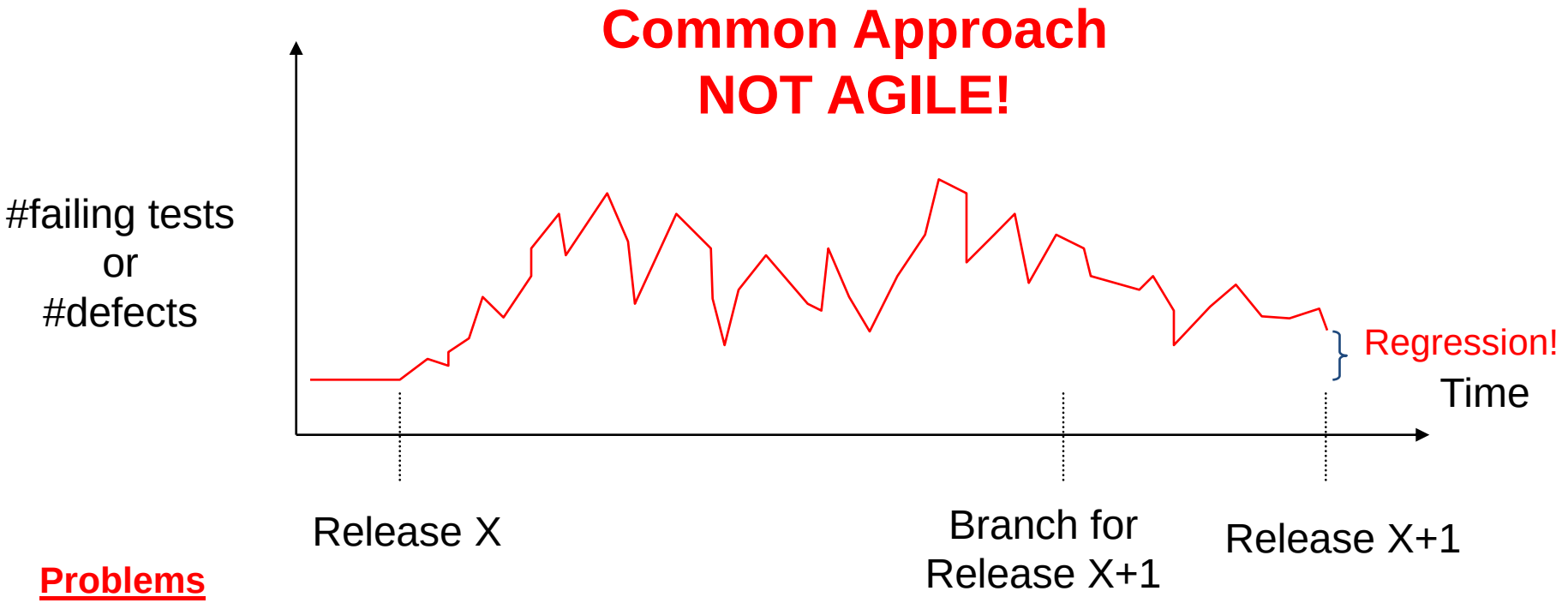
- Assuming customer validation of code is easy (i.e. linear or nonlinear algebraic equation solvers => compute the residual) …
- Can be very fast to initially create new code
- Works for the customers code right away

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!

Common Approach: Development Instability

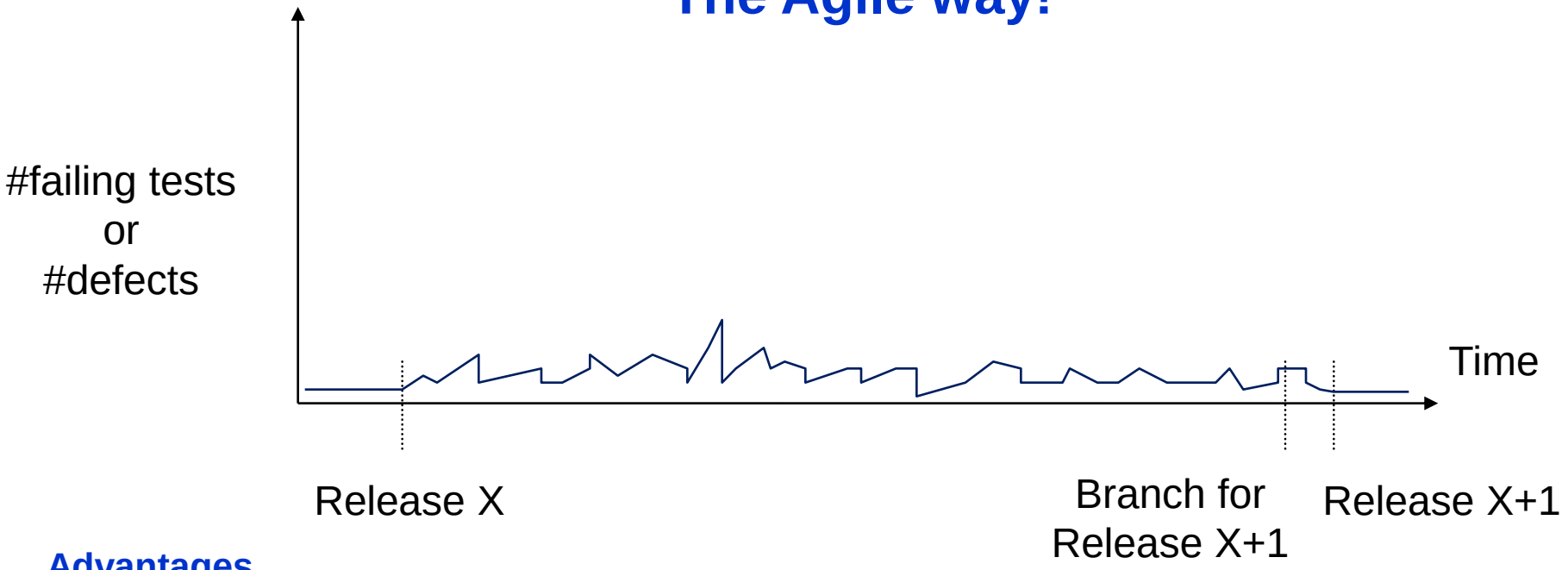


Problems

- 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

Lean/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

TriBITS: Tribal Build, Integrate, Test System

- Based on Kitware open-source toolset CMake, CTest, and Cdash developed during the adoption by Trilinos but later extended for VERA, SCALE and other projects.
- Built-in CMake-based package-arch support for partitioning a project into 'Packages' with carefully regulated dependencies with numerous features including:
  - Automatic enabling of upstream and downstream packages (critical for large projects like Trilinos, SCALE, and CASL)
  - Integrated MPI and CUDA support
  - Integrated TPL support (coordinate common TPLs across unrelated packages, common behavior for user configuration, etc.)
  - Removal of a lot of boiler-plateCMake code for creating libraries, executables, copying files, etc. …
- Powerful TRIBITS_ADD_[ADVANCED]_TEST(…) wrapper CMake functions to create advanced tests
- Integrated support for add-on repositories with add-on packages.
- TribitsCTestDriver.cmake testing driver:
  - Partitioned package-by-package output to CDash and reporting on a package-by-package basis
  - Failed packages don't propagate errors to downstream packages
  - Integrated coverage and memory testing (showing up on CDash)
  - Nightly and continuous integration (CI) test driver.
- Pre-push synchronous CI testing with the Python checkin-test.py script
- In addition: TribitsDashboardDriver system, download-cmake.py and numerous other tools

Overview of the

TriBITS Lifecycle Model

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.

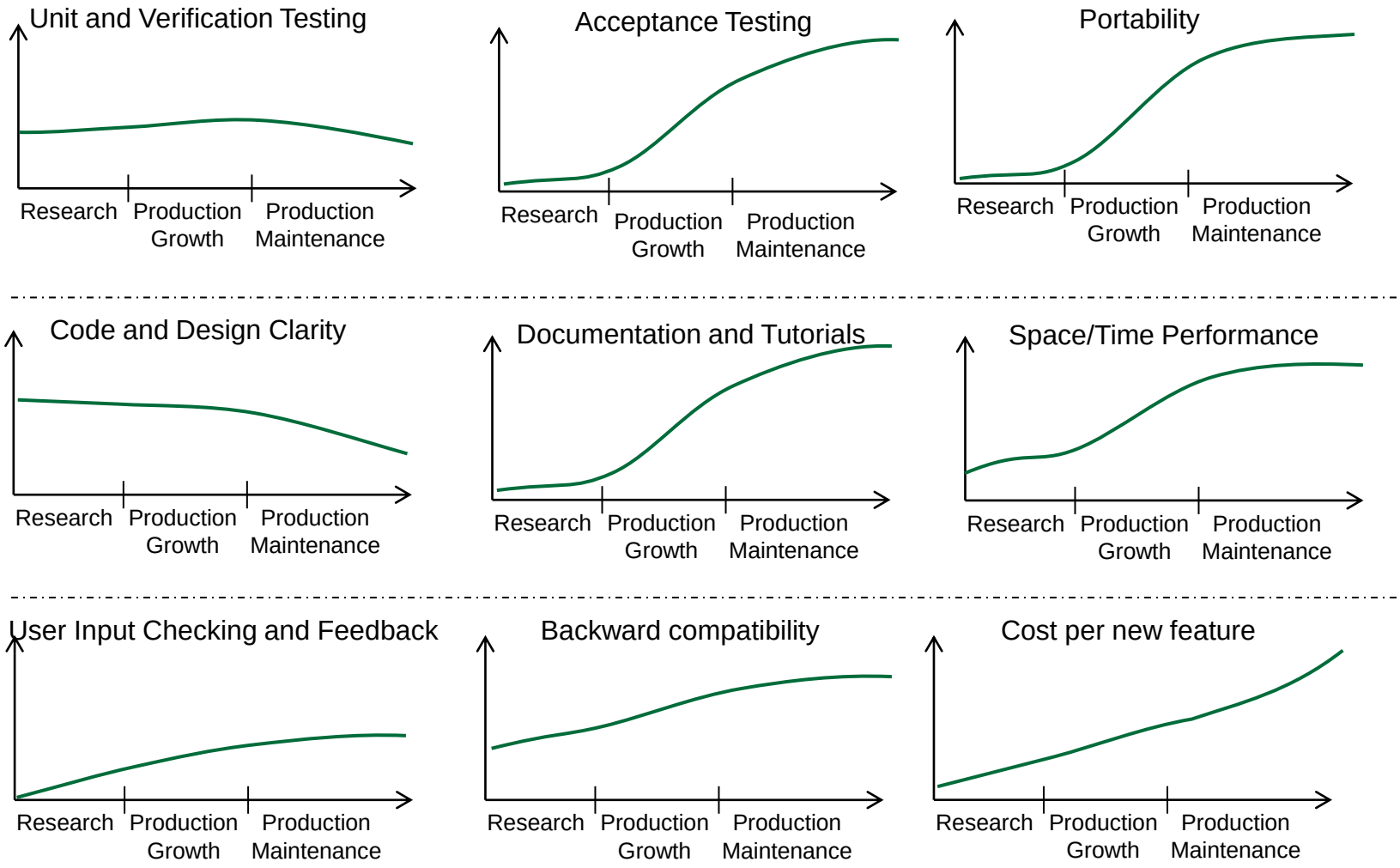
3: Production Maintenance (PM) Code

- 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”).

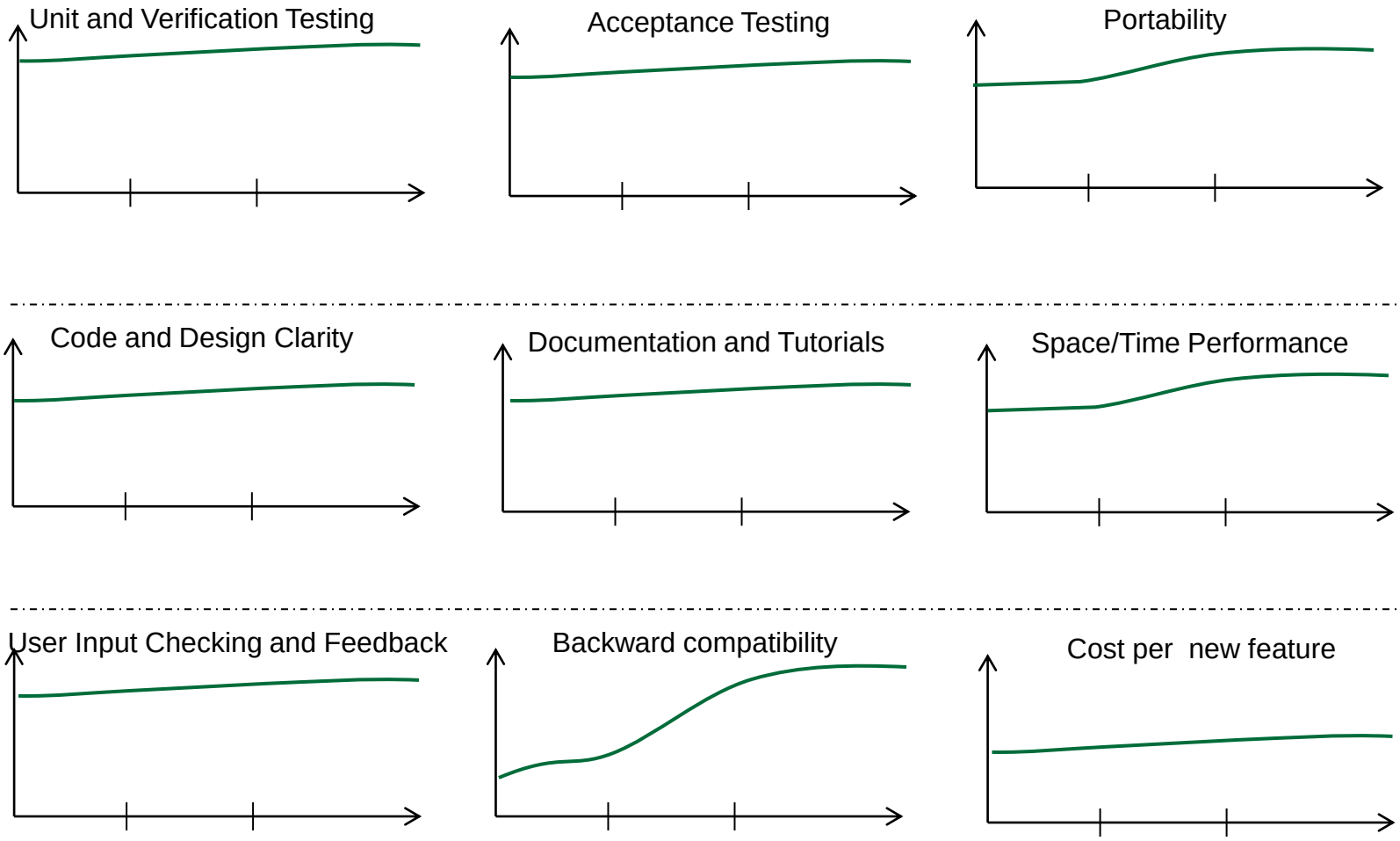
-1: Unspecified Maturity (UM) Code

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

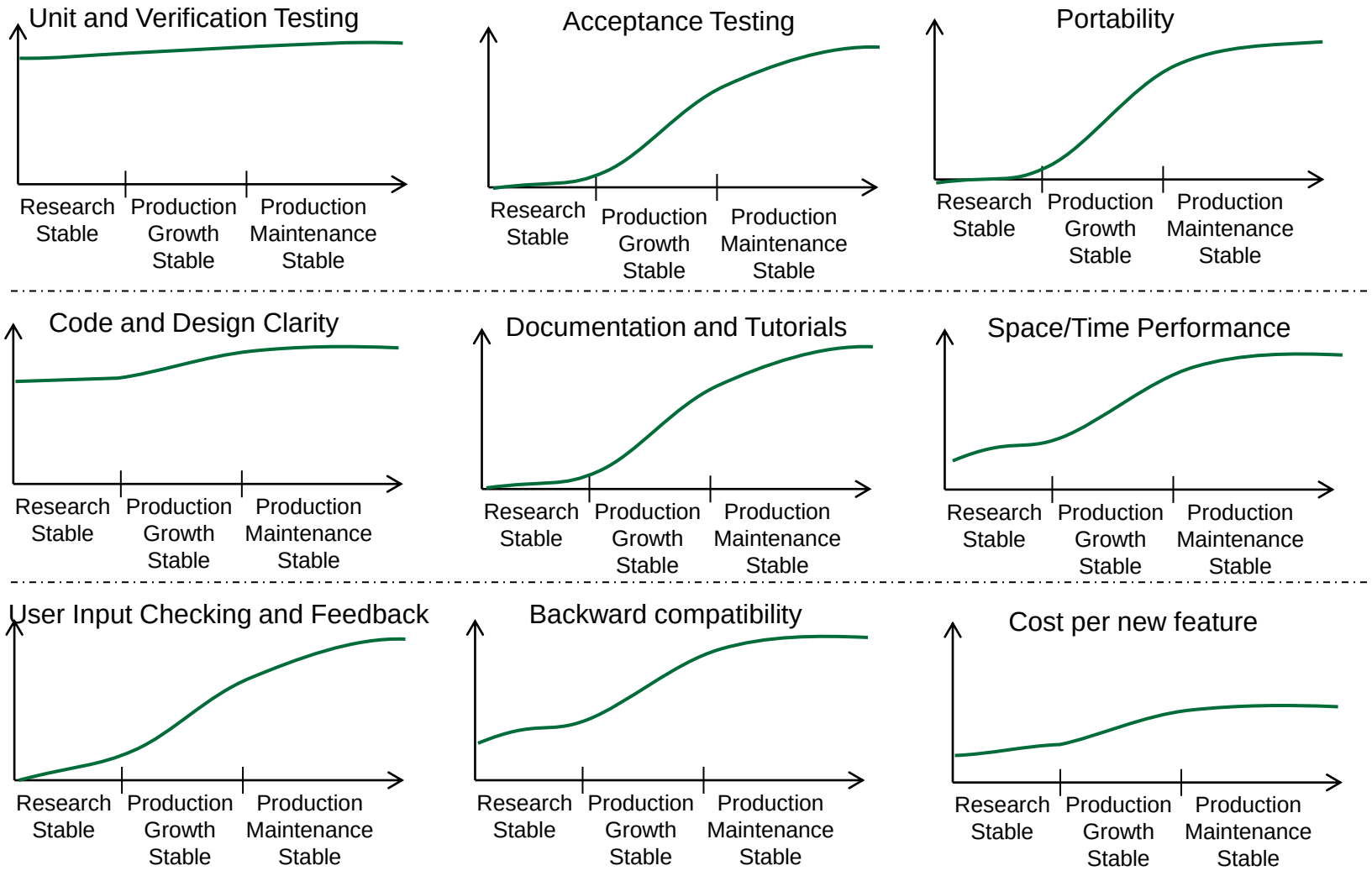
Typical (i.e. non-Lean/Agile) CSE Lifecycle



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.

Usefulness Maturity and Lifecycle Phases

- **NOTE: For research-driven software achieving "Done Done" for unproven algorithms and method is not reasonable!**
- **CSE Software should only be pushed to higher maturity levels if the software, methods, etc. have proven to be "Useful".**

Definition of "Usefulness":

- **The algorithms and methods implemented in the software have been shown to effectively address a given class of problems, and/or**
- **A given piece of software or approach makes a customer produce higher quality results, and/or**
- **Provides some other measure of value**

Corollary for Grandfathering of Legacy Software:

- **We only "Grandfather" in legacy software that has proven useful to existing customers.**

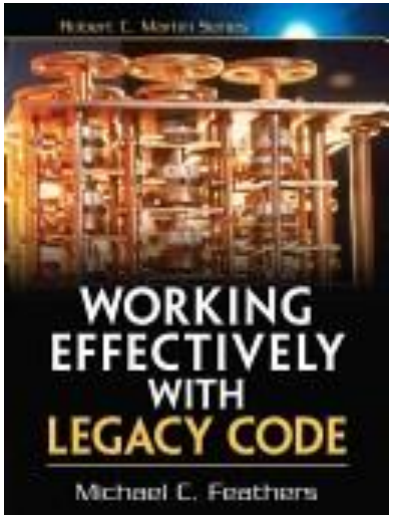
Addressing existing Legacy Software?

- **Question:** What about all the existing "Legacy" Software that we have to continue to develop and maintain? How does this lifecycle model apply to such software?

- **Answer:** Grandfather them into the TriBITS Lifecycle Model by applying 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:

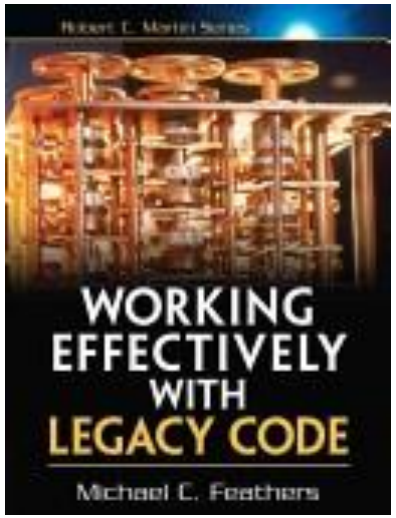
"Behavior is the most important thing about software. It is what users depend on. Users like it when we add behavior (provided it is what they really wanted), but if we change or remove behavior they depend on (introduce bugs), they stop trusting us."

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

Grandfathering of Existing Packages

Agile Legacy Software Change Algorithm:

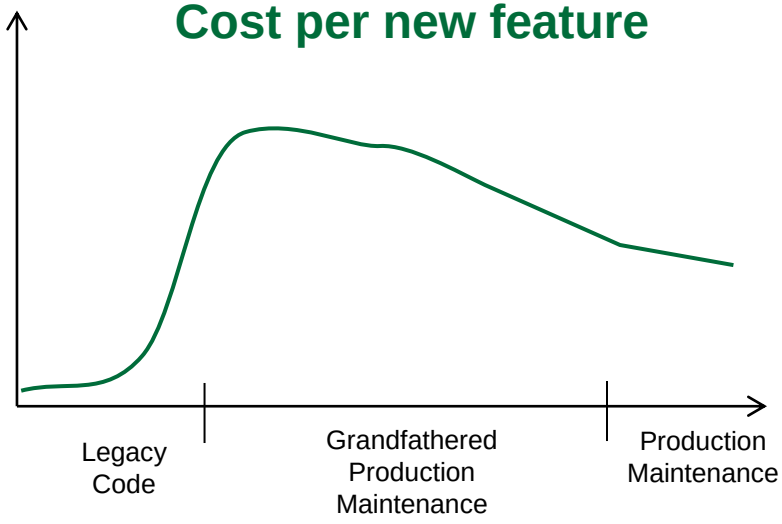
1. Identify Change Points
2. Break Dependencies
3. Cover with Unit Tests
4. Add New Functionality with Test Driven Development (TDD)
5. Refactor to removed duplication, clean up, etc.



Grandfathered Lifecycle Phases:

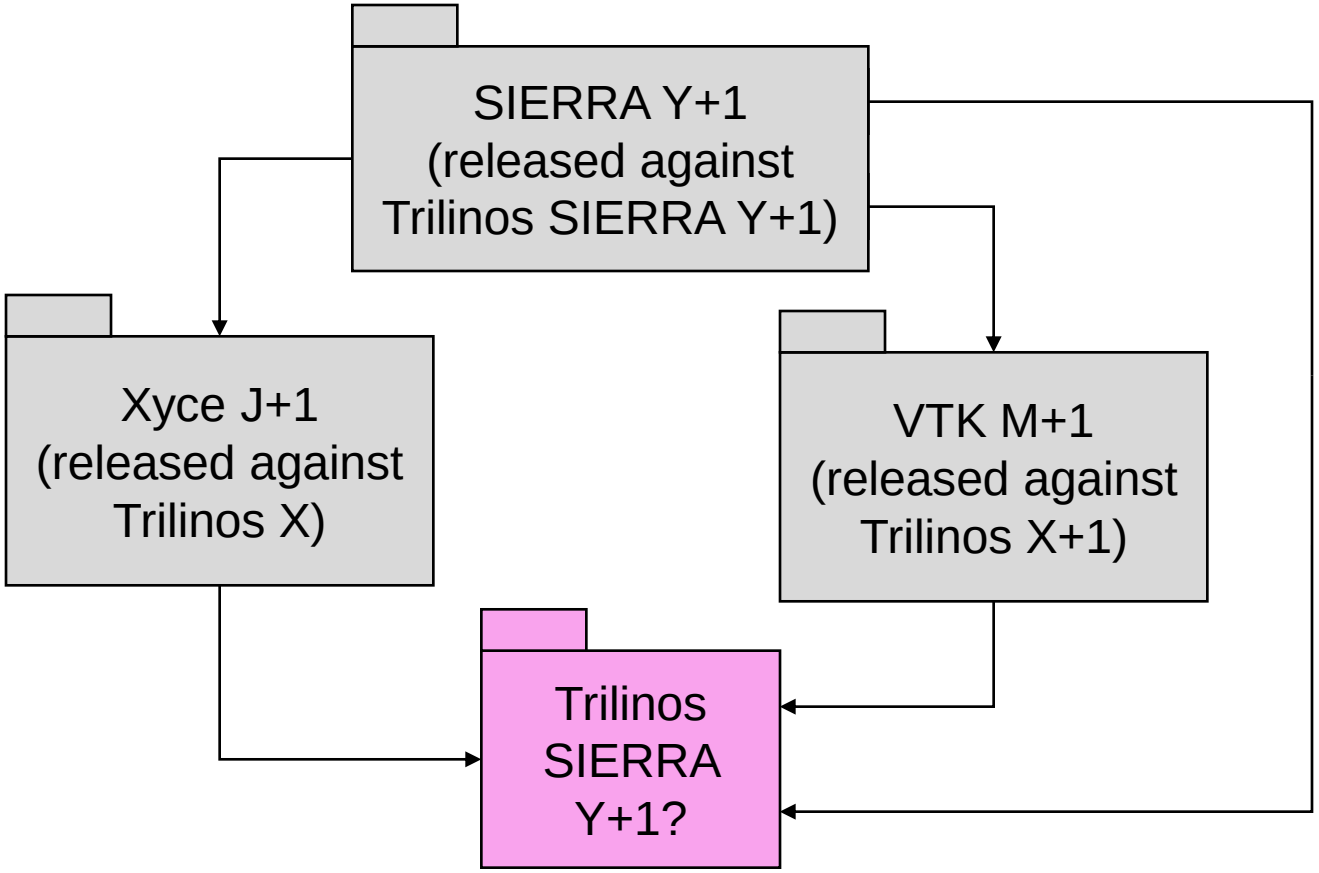
1. Grandfathered Research Stable (GRS) Code
2. Grandfathered Production Growth (GPG) Code
3. Grandfathered Production Maintenance (GPM) Code

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!



Regulated Backward Compatibility

Need for Backward Compatibility



Multiple releases of Trilinos presents a possible problem with complex applications

Solution:

=> Provide sufficient backward compatibility of Trilinos X through Trilinos SIERRA Y+1

The Cost of Maintaining Backward Compatibility

- Static interfaces and design degrades as new unanticipated functionality is added
  - Leads to messy software that is hard to understand and dangerous to modify
  - Hacks to get around design problems make the software more fragile

- Backward compatibility must be tested
  - New tests must be written and maintained by manual labor
  - Backward compatibility tests must be built and run
  - Example: Change in [TEUCHOS_]TEST_FOR_EXCEPTION(…) macros.

Bottom Line => Maintaining backward compatibility increases "technical debt"

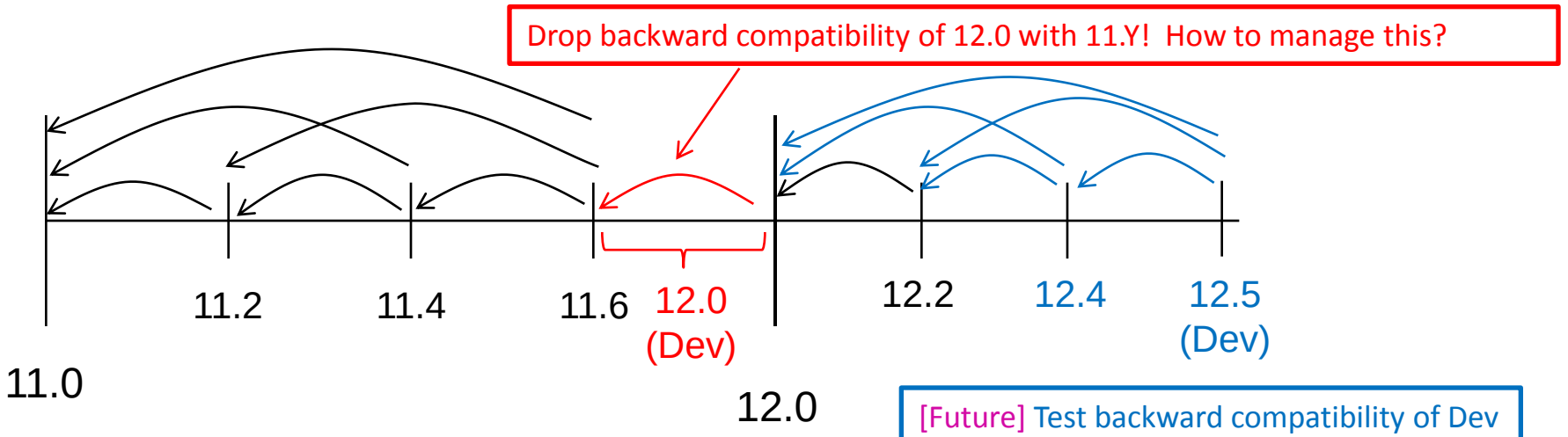
The Paradox of Backward Compatibility and Agile Development

- **Agile software development:**
  - Design changes as functionality is added and modified over multiple releases
    - Emergent Design: Design changes as the domain is better understood and functionality is added.
    - Continuous Refactoring: Keep "conceptual integrity" with clean design and clean code.
  - Requires close customer interaction and feedback to help drive software development
- **However:**
  - Many customers don't want to use software while it is constantly changing
- **The paradox:**
  - Agile developed software must change as it is developed and must have real feedback from customer use
  - Customers driving Agile developed software don't want to be constantly chasing changes.

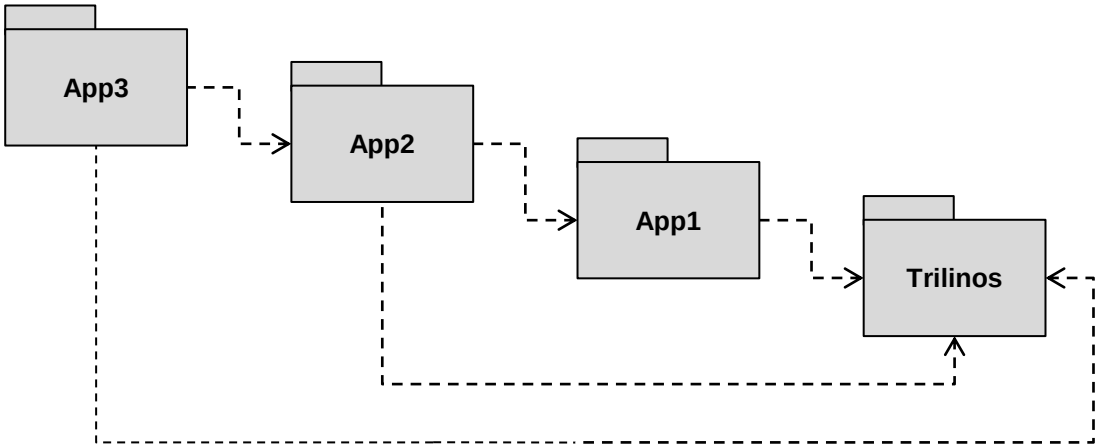
The solution => Regulated Backward Compatibility!

Regulated Backward Compatibility

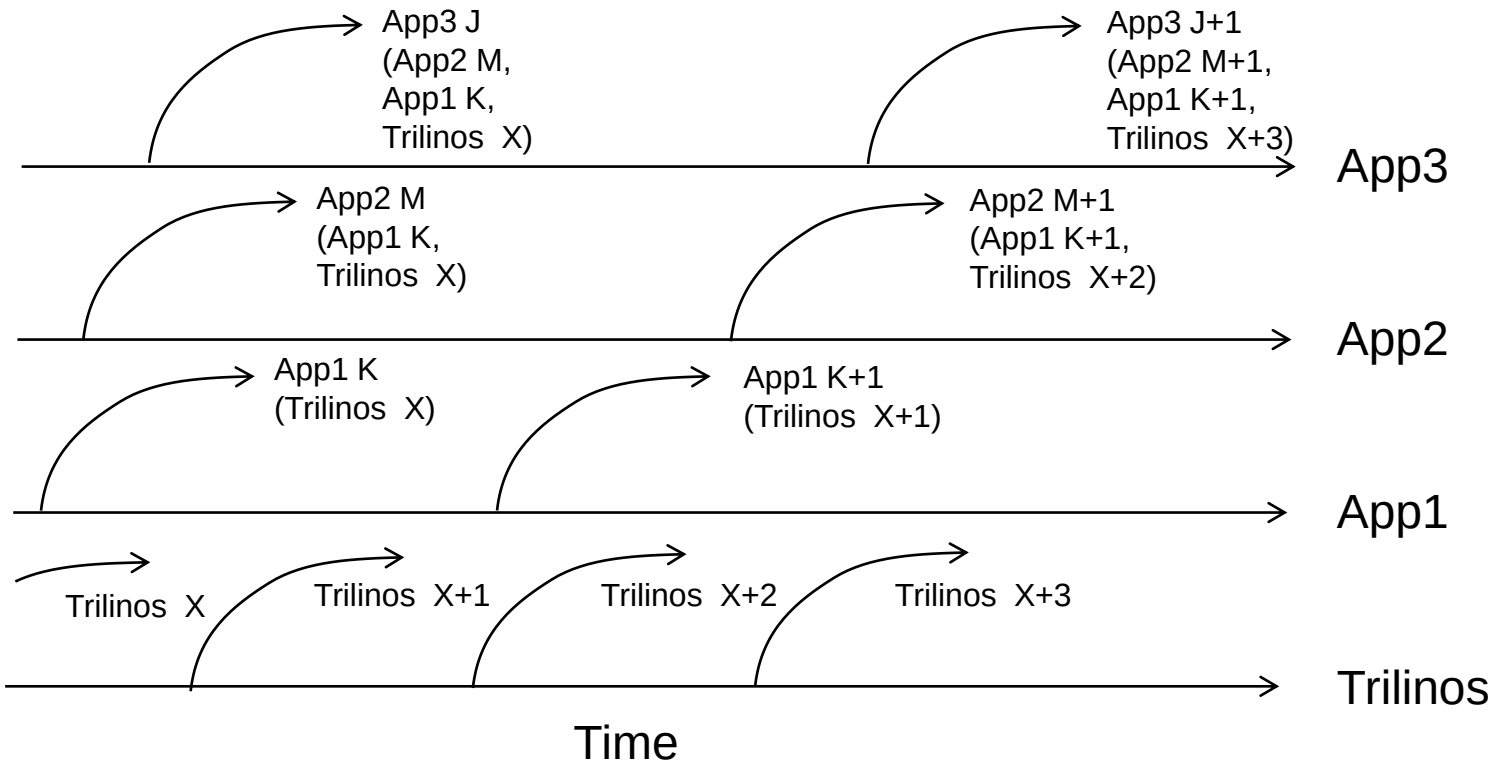
- TriBITSVersion Numbering X.Y.Z:
  - X: Defines backward compatibility set of releases
  - Y: Major release (off the master branch) number in backward compatible set
  - Z: Minor releases off the release branch X.Y
  - Y and Z: Even numbers = release, odd numbers = dev
    - Makes logic with PROJECT_version.h easier
  - Special Exception: Let X+1.0 be the development version leading to X+1.0 release
    - Allows X+1.0 to be the first major release as clear message to users
- Backward comparability between releases
  - Example: Trilinos11.6 is backward compatible with 11.0 through 11.4
  - Example: Trilinos 12.X is not compatible with Trilinos 11.Y



Three Apps that Depend on Trilinos



Releases with Three Apps and Trilinos



Regulated Backward Compatibility: Guidelines

- *Prepare users for the break in backward compatibility*:
  - Deprecate code to create compiler warnings in previous versions (see __deprecated__ attribute in GCC and Intel)
- *Fail big and hard when backward compatibility is dropped*:
  - Code should not even compile, and compile errors should be clear
  - Otherwise, code should not run at all and fail hard
  - Generate good compile or runtime error messages
- *Provide for safe straightforward upgrades*:
  - Provide sed-like scripts for making baulk changes?
    - Example: Change all class and files names 'VectorBase' to 'Vector', etc.
  - Allow namespace changes?
    - Example: TpetraNew::Vector => Tpetra::Vector for Map refactoring

Take Home Message: Plan for it and think about the user!

Deprecation Approaches

- *Deprecate classes and functions using the standard <UCPACAKGENAME> DEPRECATED macro*:

  This macro expands to the GCC deprecated attribute when Trilinos is configured with Trilinos SHOW DEPRECATED WARNINGS = ON.

- *Deprecate macros by calling dummy deprecated functions using the standard <UCPACAKGENAME> DEPRECATED macro*:

  #define OLD_MACRO_NAME(…) { SomeDeprecatedFunction(); NEW_MACRO_NAME(…) }

- *Deprecate old class names using deprecated typedefs or macro defines*:

  Preferred (nontemplates):

  typedef UCPACKAGENAME_DEPRECATED NewClassName OldClassName;

  Templated classes: #define OldClassName NewClassName;

  - Must also deprecate the header file with #warning (see below)

- *Deprecate header files by inserting protected #warning directives*:

  #ifdef __GNUC__

  # waning This header OldHeader.hpp is deprecated. Please use NewHeader.hpp

  #endif

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.

What are the Next Steps?

- Get Trilinos to adopt the TriBITS lifecycle model:
  - Feedback from a survey of Trilinos developers seems to show they support the adoption of the TriBITS lifecycle model.
- Get CASL to adopt the TriBITS Lifecycle Model:
  - Use the vocabulary of the TriBITS Lifecycle Model.
  - Encourage CASL developers to apply the Legacy Software Change Algorithm when changing existing CASL legacy codes.
  - Encourage CASL developers to use test-driven development and write clean code.
- 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).

THE END