SANDIA REPORT

SAND2012-0561
Unlimited Release
Printed February 2012

TriBITS Lifecycle Model

Version 1.0

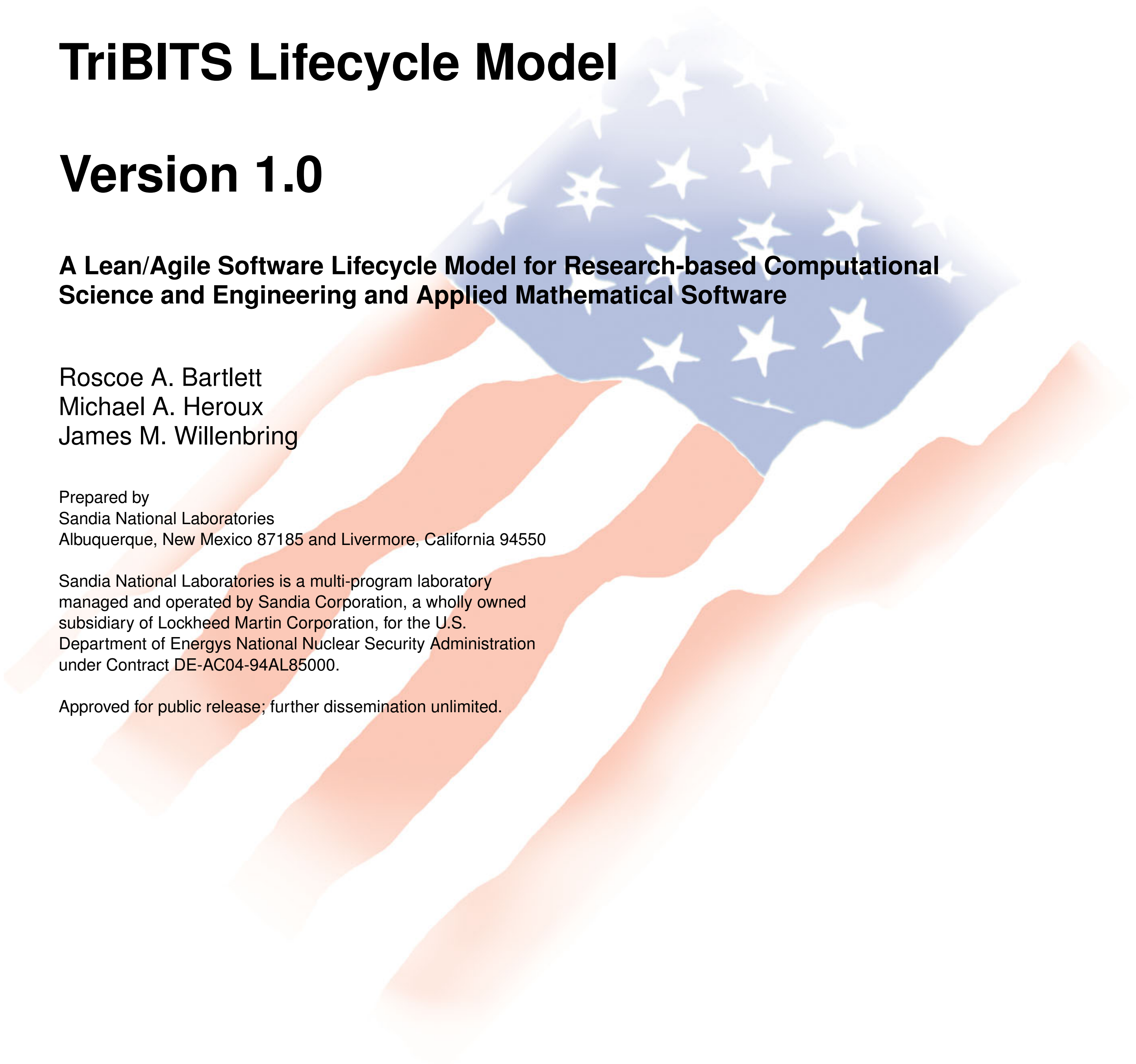
A Lean/Agile Software Lifecycle Model for Research-based Computational Science and Engineering and Applied Mathematical Software

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 Energys National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

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
Oak Ridge National Laboratory [*]
P.O. Box 2008
Oak Ridge, TN 37831

Michael A. Heroux
James M. Willenbring
Sandia National Laboratories [†]
P.O. Box 5800
Albuquerque, NM 87185-1320

Abstract

Software lifecycles are becoming an increasingly important issue for computational science & engineering (CSE) software. The process by which a piece of CSE software begins life as a set of research requirements and then matures into a trusted high-quality capability is both commonplace and extremely challenging. Although an implicit lifecycle is obviously being used in any effort, the challenges of this process–respecting the competing needs of research vs. production–cannot be overstated.

Here we describe a proposal for a well-defined software lifecycle process based on modern Lean/Agile software engineering principles. What we propose is appropriate for many CSE software projects that are initially heavily focused on research but also are expected to eventually produce usable high-quality capabilities. The model is related to TriBITS, a build, integration and testing system, which serves as a strong foundation for this lifecycle model, and aspects of this lifecycle model are ingrained in the TriBITS system.

Here, we advocate three to four phases or maturity levels that address the appropriate handling of many issues associated with the transition from research to production software.

[*]Oak Ridge National Laboratory is Managed by UT-Battelle, a partnership of the University of Tennessee and Battelle Memorial Institute, for United States Department of Energy under Contract DE-AC05-00OR22725.

[†]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 Energys National Nuclear Security Administration under Contract DE-AC04-94AL85000.

The goals of this lifecycle model are to better communicate maturity levels with customers and to help to identify and promote Software Engineering (SE) practices that will help to improve productivity and produce better software. An important collection of software in this domain is Trilinos, which is used as the motivation and the initial target for this lifecycle model. However, many other related and similar CSE (and non-CSE) software projects can also make good use of this lifecycle model, especially those that use the TriBITS system. Indeed this lifecycle process, if followed, will enable large-scale sustainable integration of many complex CSE software efforts across several institutions.

Acknowledgments

The authors would like to acknowledge the ASC and CASL programs for supporting TriBITS development.

Contents

1 Introduction and Background

Computational Science and Engineering (CSE) is experiencing a challenge in software lifecycle issues. Much software in CSE begins development as research software but at some point begins to be used in other software and it is desired (or it is expected) to eventually achieve production-quality. There is currently no sufficient software engineering lifecycle model defined for these types of CSE software that has been shown to be effective. A previous attempt to create a viable lifecycle model for CSE can be seen in the Trilinos Lifecycle Model [17]. This Trilinos lifecycle model provides for transitions of CSE software from research, to production, to maintenance (and later death). Since then we have been learning more effective techniques for software engineering. This present lifecycle model reflects what has been learned.

The goals for defining a lifecycle model for CSE (as managed by the TriBITS system for example) are many, but the most important include:

- *Allow Exploratory Research to Remain Productive*: By not requiring more practices than are necessary for doing basic research in early phases, researchers maintain maximum productivity.
- *Enable Reproducible Research*: By considering the minimal but critical software quality aspects needed for producing credible research, algorithm 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*: By focusing on the right SE practices at the right times, and the right priorities for a given phase/maturity level, developers can work more productively with as little overhead as reasonably possible.
- *Improve Production Software Quality*: By focusing on foundational issues first in early-phase development, higher-quality software will be produced as other elements of software quality are added. The end result will be production software with any required level of quality.
- *Better Communicate Maturity Levels with Customers*: By clearly defining maturity levels and advertising them well, customers and stakeholders will have the right expectations about a given piece of software, which will aid in their determination of which software to use or (at least presently) avoid using.

What is needed is a new Lean/Agile-consistent lifecycle model for research-driven CSE software that can provide a smooth transition from research to production and provide the needed level of quality for every lifecycle phase along the way. It is also necessary to appropriately communicate the current maturity level to customers and stakeholders. In order to define such a lifecycle model, it is likely that research software will need to be developed from the beginning at a better than average level of quality, in a Lean/Agile way, with good unit and verification tests. These tests must be developed at the beginning, and the internal structure of the software must be maintained at a higher quality using Agile/Emergent design and Continuous Refactoring [1]. Enhancing the quality of basic research software is needed just from the standpoint of creating credible research results that are suitable for publication in peer-reviewed journals [13, 10].

Lean methods refer to methodologies taken from Lean Product Development that have been adapted to software development [12]. *Agile methods* was coined in the early 2000's by a group of software developers in response to rigid plan-driven methods. Agile methods focus on disciplined iterative development, which is defined by a core set of values and practices [8, 14, 4]. Some of these core practices include continuous integration (CI), test-driven development (TDD), (continuous) design improvement, collective code ownership, and coding standards [8, 4]. Presently, Agile methods are considered a subset of Lean methods.

In this current document, we define a new lifecycle that will likely become the standard for many projects that use the TriBITS system (e.g. the Trilinos Project).

The rest of this document is organized as follows. Section 3 describes the current state of software engineering supported by TriBITS. This is done to provide a frame of context for lifecycle discussions and to avoid having to restate the same foundations several times scattered throughout the document. Section 4 then gives a short overview of the new TriBITS lifecycle model and some short discussion and motivation. The concept of *self-sustaining software*, which is the foundation of the TriBITS lifecycle model, is defined in Section 5. This is followed in Section 6 with a detailed motivation and discussion of regulated backward compatibility. With all of that background in place, a more detailed prose discussion of the TriBITS lifecycle model phases is given in Section 7. The difference between and relationship of software engineering maturity and software usefulness maturity is the topic of Section 8. The important topic of risk analysis and the role of acceptance testing is considered in Section 9. This is followed by the comparison of the new TriBITS lifecycle model to a typical lifecycle model used by a CSE project in Section 10 . Since any lifecycle model that does not acknowledge the current state of a project is never going to be followed, a discussion for the grandfathering of existing packages is presented in Section 11. The TriBITS lifecycle model is summarized and next steps are given in Section 12. Appendix A compares the proposed TriBITS lifecycle model with the Trilinos lifecycle 1.0 model. Appendix B discusses some details about how the TriBITS software system can help to support the TriBITS lifecycle model.

2 Common Implicit Model: A Validation-Centric Approach

Many if not most CSE projects have no formal software lifecycle model. At the same time, these projects perform the activities required to develop software: elicit and analyze requirements, design and implement software, test and maintain the product. Therefore, although implicit, each project has some kind of lifecycle model. In our experience, the most common implicit CSE lifecycle model can be described as a *validation-centric approach* (VCA).

Roughly speaking, validation is *doing the right thing* (whereas verification is *doing things right*). Validation is viewing the software product as a black box that is supposed to provide a certain behavior and functionality. Validation is not concerned about the internal structure of the product.

VCA can be a very attractive approach when validation is easy to perform. For example, testing the correct behavior and efficiency of a nonlinear or linear solver is very straightforward. If a solver returns a solution, it is easy to compute the execution time and residual norm as part of using the solver. Even if the solver does not behave as the developer intended (verification), and may not be implemented optimally, there is little risk of an undetected failure, and performance degradations are easily detected. If an application has several solvers to pick from, then even the impact of a software regression in one solver is mitigated by being able to switch to another.

When VCA works, it introduces very little overhead to the software engineering process, especially since validation is an essential part of product certification. In fact, if a software product is being developed for a specific customer, validation is often occurring as part of the development process,

Side Note: *The Trilinos Project* is an effort to facilitate the design, development, integration and on-going support of foundational libraries for computational science and engineering applications. Trilinos is composed of a set of different packages with a wide range of capabilities including basic distributed memory linear algebra, load balancing, discretization support, and wide variety of nonlinear analysis methods and much more. Trilinos packages have complex dependencies on each other and create a challenging software development environment. The primary programming language for Trilinos is C++ and therefore C++ considerations dominate Trilinos development.

since the customer is using the product as it is being developed. Furthermore, investment in product-specific unit and integration tests is expensive. If the future of a software product is uncertain, a development team may not want to dedicate resources to these activities. VCA is often (but not always) the lowest cost way to initially assure correct software behavior.

From the above discussion, it is clear why VCA may seem attractive. However, despite this attraction, VCA is an ineffective long-term approach for all but the most easily validated software products. This become particularly apparent as a product matures and refactorings are made to address requirements such as feature requests and porting to new platforms. Numerous studies indicate that maintenance can be as much as 75% or more of the total cost of a successful software product [9]. VCA is initially inexpensive, but ultimately costs much more. Furthermore, in many cases it leads to early abandonment of a product simply because refactoring it become untenable. Some of the specific challenges associated with VCA are:

1. Feature expansion beyond validated functionality: As a software product becomes widely used, feature are often added that are not covered by validation testing. These feature are immediately used by some customer, but the customer's code is not added to the validation suite. Any such code is at risk when performing refactoring.
2. Loss of validation capabilities: Although a customer application may initially be used to validate a product, relationships with that customer may change and the validation process will break down.
3. Inability to confidently refactor: In general, without internal testing in the form of easily checked unit tests and product-specific integrated testing, refactoring efforts cannot proceed incrementally. Furthermore, running validation tests for incremental changes is often cumbersome and time-consuming, reducing the efficiency of the refactoring process.

Although validation-centric models are commonly used in CSE, more effective approaches are needed. This comes at a cost: teams must have the resources, tools and training to effectively develop sustainable software. In the remainder of this paper we present approaches we have found effective.

3 Current Baseline State for TriBITS projects

Before describing the new Tribits lifecycle model we provide some background on the current state of TriBITS-based software engineering and development practices (which is used by the Trilinos project, for exmaple) to set the foundation on which this lifecycle model will be based.

These practices and policies are ingrained into a TriBITS development environment and are taken for granted in this discussion of the new TriBITS lifecycle process.

The short list of relevant core TriBITS software engineering processes and practices are:

- Official separation of TriBITS project code into Primary Stable (PS), Secondary Stable (SS), and Experimental (EX) for testing purposes (see definition of PS, SS, and EX in text box below)
- Partitioning of software into different packages (and subpackages) with carefully controlled and managed dependencies between the pieces of software
- Synchronous (pre-push) CI testing of PS tested code using the Python tool checkin-test.py
- Asynchronous (post-push) CI testing of PS and SS tested code using package-based CTest driver posting results to CDash[1]
- Increasingly more extensive automated nightly regression testing and portability testing (on a growing list of platforms) of PS and SS tested code posting results to CDash
- Strong compiler warnings enabled with g++ flags such `-ansi -pedantic -Wall -Wshadow -Woverloaded-virtual`
- strong policies on the maintenance of 100% clean PS and SS builds and tests for all Nightly and CI builds

The official segregation of (Primary and Secondary) Stable tested code from Experimental code allows the development team to define and enforce fairly rigorous policies on keeping Stable code building and having all the tests of Stable features run successfully. This is maintained through the synchronous (pre-push) CI and asynchronous (post-poss) CI testing tools and processes and strong peer pressure to keep PS and SS code builds and tests 100% clean. However, with respect to lifecycle, having 100% clean tests means very little if test coverage is low (which is the case for many existing packages). Technically speaking, Stable code with test coverage not near 100% at all times means the code is not meeting basic Agile development standards (see [4] and [9]). The distinction between PS, SS, and EX code in the TriBITS system has to do with the commitment of the development team to maintain (or not, in the EX case) the status of builds and passing tests on various platforms and has nothing to do with the assumed quality of the software. Official assignments and metrics of software quality in projects like Trilinos are currently non-existent (but this current document is trying to change that).

4 Overview of the TriBITS Software Lifecycle Model

Here we propose a lifecycle model with four possible phases or maturity levels (these can be thought of as either phases or maturity levels depending on usage and frame of reference). The concepts of *self-sustaining software* and *regulated backward compatibility*, which are critical in the definition and goals of these lifecycle phases, are described in Section 5 and Section 6, respectively. However, before going into detail about these concepts and a more detailed discussion of these lifecycle phases in Section 7, we first preview these phases here.

The proposed lifecycle phases / maturity levels / classifications are:

[1]CDash (www.cdash.org) is an open source, web-based software testing server. CDash aggregates, analyzes and displays the results of software testing processes submitted from clients located around the world. Developers depend on CDash to convey the state of a software system, and to continually improve its quality. CDash is a part of a larger software process that integrates Kitware's CMake, CTest, and CPack tools, as well as other external packages used to design, manage and maintain large-scale software systems.

Defined: PS, SS, and EX tested code:
PS (Primary Stable) tested code has minimal outside dependencies (currently, PS code in can only maximally have required dependencies on the external TPLs BLAS, LAPACK, and MPI) and represents critical functionality such that if broken would hamper core development efforts for many developers.
SS (Secondary Stable) tested code has dependencies on more than just the minimal set of TPLs (such as depending on Boost) or is code that does not represent critical functionality such that if broken would significantly hamper the work of other developers. Most SS code is tested by the asynchronous CI server for the primary development platform after commits are pushed to the main development repository. SS code is also tested nightly on many platforms.
EX (Experimental) (non-)tested code is not PS or SS code and not tested in any official testing process.

1. Exploratory (EP):
   - Primary purpose is to explore alternative approaches and prototypes, not to create software.
   - Generally <u>not</u> developed in a Lean/Agile consistent way.
   - Does not provide sufficient unit (or otherwise) testing to demonstrate correctness.
   - Could be categorized as SS code with respect to CI and nightly testing, but in general would be considered to be untested EX code.
   - 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 likely still be allowed).
   - Generally should <u>not</u> go out in open releases (but could go out in releases and is allowed by this lifecycle model).
   - Does <u>not</u> provide a direct foundation for creating production-quality code and should be put to the side or thrown away when starting product development.
2. Research Stable (RS):
   - 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). This does not necessarily need to have a much higher initial cost (see side-note on Page 20) but it may be big change for a typical programmer.
   - Could be treated as PS or SS code with respect to testing.
   - Has a very clean design and code base maintained through Agile practices of emergent design and constant refactoring.
   - Generally does <u>not</u> 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.
   - <u>Is</u> appropriate to be used only by "expert" users.
   - <u>Is</u> appropriate to be used only in "friendly" customer codes.
   - Generally should <u>not</u> 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.

3. Production Growth (PG):
    - 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.
4. Production Maintenance (PM):
    - 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").
5. Unspecified Maturity (UM):
    - Provides no official indication of maturity or quality.

Figure 1 shows how the different aspects of software quality and maturity should progress over the phases from Research Stable through Production Maintenance. What is shown is that from the very beginning, Lean/Agile research software has a high level of unit and verification testing (because it is developed using TDD) and maintains very a clean simple design and code base that is only improved over time. Fundamental testing and code clarity lie at the foundation of self-sustaining software (Section 5) and for the transition to eventual production quality. Acceptance testing is related to specific end-user applications and would naturally start out at a low level and increase as more users accept the software and add acceptance tests. Note that classical validation testing would also be categorized as acceptance testing here; more on that in Section 9.

The more user-focused quality aspects of production software which include documentation and tutorials, user input checking and feedback, backwards compatibility, portability, and space/time performance are improved as needed by end users and justified by various demands. The level of these user-oriented aspects of quality can be in very different stages of maturity yet the software can still be considered very good Lean/Agile software. What differentiates Lean/Agile Research software from Lean/Agile Production-quality software is not the amount of testing and proof of correctness but instead is the level of quality in these more user-focused areas. However, when using basic Agile development practices (including Emergent Design and Continuous Refactoring), even "research" software will tend to have a clean internal structure and have reasonably consistent user interfaces. The more user-oriented aspects of production software will not be overlooked as the software becomes a product because they are what users most directly see. However, the core foundation of the software that makes it self-sustaining, which includes fundamental testing and code/design simplicity and clarity, is not directly seen by users, and these critical aspects often get overlooked as software is "hardened" toward production and result in non-sustainable software. Therefore, the foundational elements of strong testing and clean

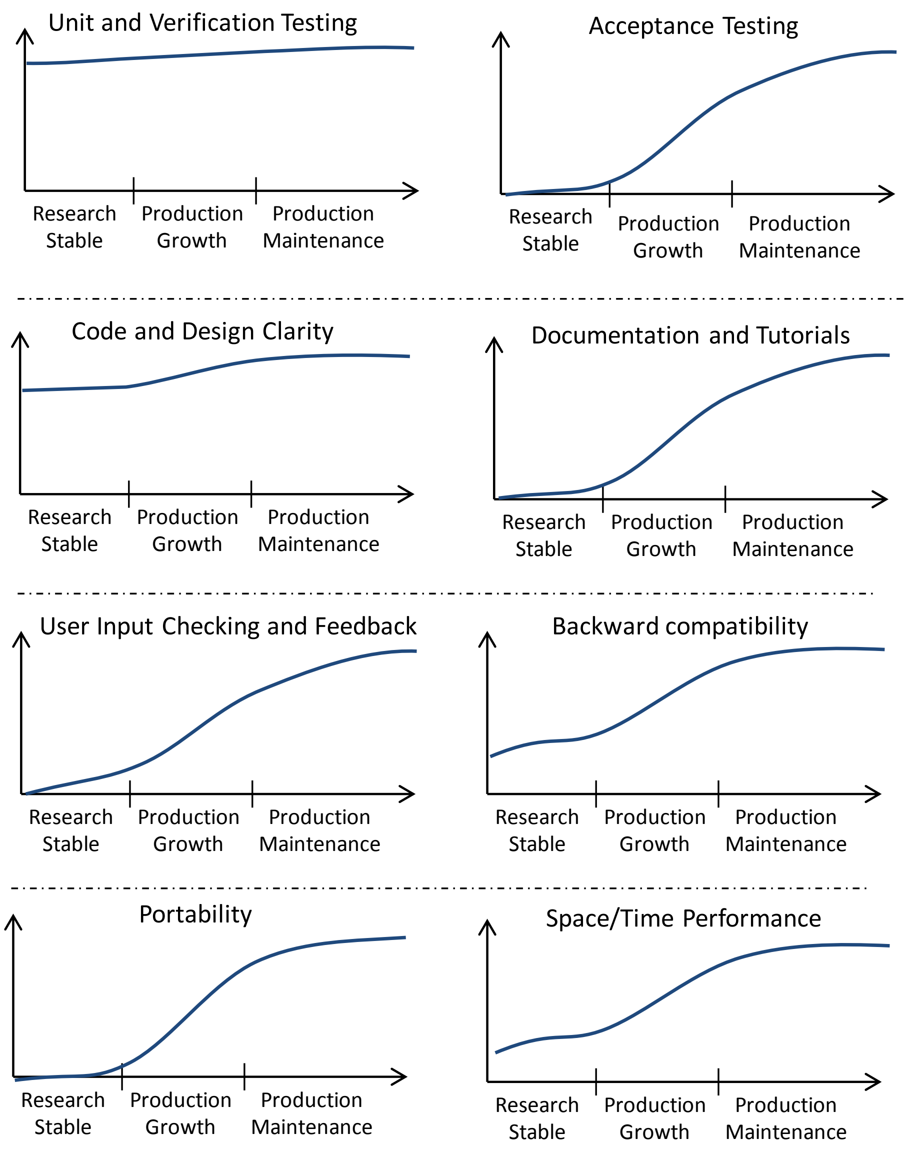


Figure 1. Typical levels of various production quality metrics in the different phases of the proposed Lean/Agile-consistent TriBITS lifecycle model.

design and code must be established at the beginning and maintained throughout the development of the software, even from the first lines of research code.

It is worth mentioning that issues like version-control, automated testing, nightly and continuous integration testing, and other basic extremely well accepted Agile software technical practices are already intimately woven into a TriBITS-based project (see Section 3). However, while these policies and practices are very important, they do not directly impact a lifecycle model except to make such a lifecycle model easier and safer to implement. Therefore, we will not discuss such basic software practices and policies in this document except where needed, and instead refer the reader to references for details (again, see Section 3).

What is important to grasp about this new TriBITS lifecycle model is that steady progress is made in each of the areas (and others) shown in Figure 1. There are no abrupt transitional events that are required to move from one stage to the next. When enough of the core production quality aspects are in place, the development team can simply declare a package to be in a higher (or lower) maturity level or stage. The level of quality in any given area or set of areas needed to achieve the next higher maturity level is somewhat subjective and will need to be quantified or otherwise evaluated in some way when implementing this lifecycle model in a particular project.

The Exploratory code phase mentioned above is for software that is only used to quickly experiment with different approaches. The Exploratory code phase really should not exist in a Lean/Agile consistent lifecycle model, but is needed as a catch-all for code that lacks the level of unit and verification testing or level of design and code clarity needed to be considered consistent with Lean/Agile software, and therefore cannot be considered Research Stable code. Such exploratory prototyping code should be discarded and rewritten from scratch when entering the Research Stable phase. Also, as mentioned above, Exploratory code should likely not be used even for peer-reviewed published research for journals that require reproducible research. Code intended for publishing research results should be written from the beginning as Lean/Agile consistent Research Stable code.

The Unspecified Maturity code phase is used to classify software that the development teams choose to opt out of the TriBITS lifecycle model.

Before describing the four different phases of this new TriBITS lifecycle model in more detail, the important concepts of self-sustaining software and regulated backward compatibility are defined in the next two sections.

5 Self-Sustaining Open-Source Software: Defined

The CSE domain is complex and challenging for developing and sustaining high-quality software. Many CSE projects have development and usage lifetimes that span 10 to 30 years or more. These projects have to port to many different platforms over their lifetime as computing technology shifts, and the software must be augmented and changed as new algorithms are developed [16]. In such an environment, creating a strong dependence on commercial tools and libraries can be a large risk. Companies are purchased and product lines go away. (For example, Intel purchased the KAI C++ compiler in the late 1990's and removed it from the market. It took years before the Intel compilers reached the same level of quality as the KAI C++ compiler in compiling and optimizing deeply templated C++ code.) In addition, complex non-commercial software produced in the U.S. national laboratories and universities, which require continuing development, maintenance and support teams also represent a risk to long-lived CSE projects. For example, what happens to customer projects that adopt a complex SciDAC software library when the funding goes away and removes the associated supporting SciDAC-funded development and support team?

We advocate that CSE software (such as Trilinos and related software packages) intended for use by other CSE projects should be *self-sustaining*, so that customer project teams could take over the basic maintenance and support of the software for their own use, if needed.

We define *Self-Sustaining Software* to have the following properties:

- *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 [6].
- *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).

The software must have a sufficiently free open-source license so that customer projects can make needed changes to the software to suit their needs and do critical porting work. Alternatively, customers of non-open-source commercial software must rely on the supplying vendor to make needed modifications and porting which they might not be able to do for various reasons. Also, the open-source license must be open enough to allow broad reuse. For example, the GNU Lesser General Public License (LGPL) is acceptable for many organizations, but some organizations (such a commercial and even some private for-profit companies) cannot use LGPL software because LGPL enforces a copylefting principle such that any changes made to the acquired software must also be open source. When integrating LGPL software into a client software package it can be difficult to determine the boundary between the two. Rather than risking legal issues these companies avoid LGPL software. GPL software is also inappropriate for self-sustaining CSE software by any reasonable measure. However, the BSD license and similar licenses seem to be open enough to allow no restrictions on use and reuse, and should therefore be preferred as the default open-source license for self-sustaining software.

A high-level document is needed to define the scope and the high-level goals and core domain of the software which is needed to maintain the "conceptual integrity" of the software [5]. This document will be short and focused and will define the high-level "Core Domain Model" for the particular piece of CSE software [6]. A good example of a domain model for a multi-physics coupling package is given in [11].

Strong unit and verification tests are at least as important as a high-level domain-model document. Such tests are critical for safe and efficient changes such as adding new features, porting to new platforms, and generally refactoring of the code as needed, without breaking behavior or destroying backward compatibility (see [7]). Of course, code with strong unit and verification tests will have very few defects (i.e., in Agile, there are no defects, only missing tests). The testing environment must be an integral part of the development process from the very beginning (preferably using Test-Drive Development (TDD) [3]). Any software lifecycle model and process that does not include strong unit and verification testing from the very beginning of the project is not Lean/Agile consistent and, practically speaking, cannot be used as the foundation for eventually producing trusted production-quality software. This is probably the biggest weakness of the commonly used validation-centric model described in Section 2.

Maintaining a clean and simple internal structure of the code is crucial for achieving self-sustaining software. No matter how good the unit and verification tests are, if the code is too convoluted and/or has too many entangling dependencies and tricky behaviors, the software will not be able to be avoidably maintained, ported to new platforms, or changed for other purposes. Developing software which keeps a clean internal structure is most efficiently done using continuous refactoring and redesign [4]. Without applying such a skilled and rigorous process for maintaining the software, under maintenance and further development it will eventually die the slow death of "software entropy" and become unsustainable [5]. Developing software of this type requires a high degree of skill and knowledge and places a very high standard on CSE development teams which first create and later expand and maintain the software.

A self-sustaining software package must also have carefully managed dependencies within itself and with its upstream software dependencies. Within the package, if it has too many entangling dependencies and if any particular customer does not need the majority of functionality provided by the software, then the customer is at greater risk because they may be forced to build (and possibly port) a lot of software they don't actually use. For example, a given downstream customer may only fundamentally need a few classes but if the given software has entangling dependencies within itself, the customer may be forced to port hundreds of thousands of lines of code just to get functionality that should be contained in a few thousand lines of code. Similar to entangling dependencies within a given piece of software, the more external upstream software that is required to be downloaded, configured, and built, the greater the risk. In extreme cases, the volume of external software package installation required before building the desired package can greatly complicate the installation and porting of the software. However, the goal is not to have zero dependencies. Therefore, self-sustaining software must carefully manage internal and external dependencies.

Also, it is not enough for a given piece of software to satisfy the basic properties of self-sustaining software, but it is also critical that the upstream dependencies of self-sustaining software also themselves be self-sustaining software. Therefore, the definition of self-sustaining software is recursive in nature. For example, suppose a piece of software is clear and well tested but has a critical dependency on a commercial numerical library that is unique and not easy to replace. If the vendor removes support for this critical commercial numerical library, the downstream open-source software may become unusable and non-portable to future platforms.

Any software that does not maintain these properties as it is ported and maintained will eventually become unsustainable, which then becomes a liability to its various downstream customer CSE software projects. Again, this is a high bar to define for CSE software developers but it critical for sustainable large-scale CSE software integration.

Note that self-sustaining software does not necessarily have good user documentation, or any user-oriented examples, nor necessarily produces good error messages for invalid user input. While these properties of any piece of software are important for the adoption and usage of software they do not affect the sustainability of the software by an existing set of client projects. The tests define the expected behavior of the software that must be maintained (or purposefully changed by changing the tests).

6 Regulated Backward Compatibility

Backward compatibility is a central issue in Agile software development. The issue of regulated backward compatibility was mentioned in Section 4 and is a critical aspect of CSE software intended for use by other projects. First we discuss the paradox of backward compatibility in Agile software development, and then we present the TriBITS strategy for managing regulated backward compatibility.

6.1 The Paradox of Backward Compatibility and Constantly Changing Agile Software

Agile software is iteratively developed and released to customers with very early releases, preferably containing very little initial functionality, followed by many frequent releases with increasing functionality [8]. All the while, the software is constantly being refactored and refined with each new release. Software developed in an Agile way needs early and frequent feedback from real customers. Without that early and frequent feedback from real customer usage, Agile methods fundamentally fail.

However, most prospective customers are not very interested in starting to adopt and use a piece of software that is constantly changing with every new iterative release. Interface changes in aggressively developed software break customer code and require the downstream developers to constantly be chasing changes, and dealing with broken builds and changing behavior. Most customers would rather wait until the software stabilizes before deciding to start using the software (and therefore start giving feedback). But herein lies the paradox: Agile developed software cannot stabilize its interfaces and behavior unless customers use the software and give feedback after many iterations of releases, but customers do not want to adopt software that is constantly changing and therefore are not providing any feedback.

So if customers do not want to use (and therefore provide feedback for) software that is constantly changing and Agile developed software cannot stabilize unless its gets feedback from customer usage, how can Agile development work? How can we develop Agile software without placing an unreasonable burden on customers? The answer is to carefully manage changes in Agile developed software interfaces and behavior and smooth the transition for customers to make upgrades of iterative (or continuous) releases as easy, safe, and painless as possible. The way we do that is to adopt a rigorous set of policies and practices for *regulated backward compatibility* (to be defined in Section 6.4).

6.2 The Need for Backward Compatibility

Backward compatibility is critical for safe upgrades to new releases of software and for the composability and compatibility of different software collections. If every new release of a piece of software breaks backward compatibility, it discourages users from depending on the software and/or discourages accepting new releases. In the case of CI, if a package is constantly breaking backward compatibility, then downstream packages must constantly be updated to keep up, making for a difficult and inefficient development process. In the case of composability of software through releases, some backward compatibility is critical to allow for different collections of software to be integrated.

To demonstrate the critical need for backward compatibility for allowing the composability of software, consider the scenario shown in Figure 2. This scenario shows three different software collections that depend on Trilinos and each other. Here, Xyce version J+1 is released which is tested and certified against Trilinos version X. After that, Trilinos puts out release X+1. This is followed by the release of VTK version M+1, which is tested and certified against Trilinos X+1. Finally, SIERRA version Y+1 is released which is tested and certified against its own local snapshot of Trilinos called Trilinos version SIERRA Y+1. Here SIERRA has optional dependencies on Xyce and VTK for which it uses static releases. In this scenario, what version of Trilinos can be used that is compatible with Xyce J+1, VTK M+1 and SIERRA Y+1? If each Trilinos release breaks backward compatibility with each previous release in a non-trivial way, then there is no single version of Trilinos in existence that will allow the integration of Xyce J+1, VTK M+1 and SIERRA Y+1 in the same statically linked executable and trying to mix in multiple library versions of Trilinos will result in link-time errors[2].

However, if Trilinos releases maintain rigorous backward compatibility between releases Trilinos X through Trilinos SIERRA Y+1, then the integration and release scenario shown in Figure 2 is easily achieved.

Without sufficient backward compatibility, such integration scenarios becomes very difficult and would otherwise require that the (static) releases of Xyce J+1 and VTK M+1 be updated to Trilinos version SIERRA Y+1. Such upgrades would be done, typically, without the aid of the core Xyce or VTK development teams and performing these upgrades could be expensive and risky. All of these difficulties, risks, cost, and uncertainty in developing and releasing such downstream software all but go away if the upstream library (Trilinos in this case) maintains backward compatibility across all of the needed versions.

6.3 The Cost of Maintaining Backward Compatibility

While maintaining some backward compatibility is a critical requirement for large-scale development and software integration (in CI or through releases) it comes with potentially high costs over a long-lived project. The main costs are (i) the extra testing needed to ensure backward compatibility, (ii) the costs of refactoring code that must maintain an out-of-date interface and (iii) restrictions that prohibit better or safer designs and code.

[2]When using dynamic linking, it is possible to mix multiple versions of a library in the same executable so it is possible to use non-backward compatible versions of a library in a dynamically linked program. However, may of the high-end HPC platforms only allow static linking so relying on dynamic linking to address incompatible library versions is not a viable solution for CSE HPC software. Furthermore, dynamic linking makes rigorous validation of an executable program risky since execution can affected by a simple change in the runtime environment

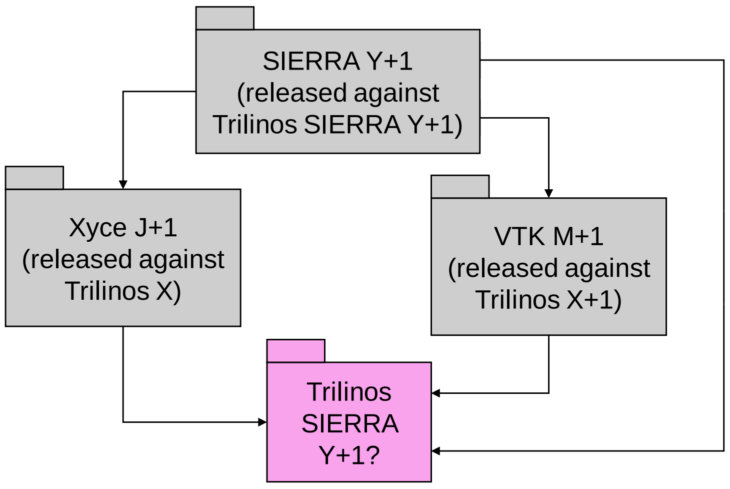


Figure 2. The need for backward compatibility among multiple releases of Trilinos and customer applications: If Trilinos X, X+1 and SIERRA Y+1 have no common compatible core, there is no version of Trilinos that can provide the necessary functionality for these three composed applications.

As new features are developed and new interfaces are created, maintaining backward compatibility requires keeping older versions of classes and functions around with the old behavior. Keeping old interfaces and functionality bloats the code base and requires more code to build and more tests to run and maintain. This overhead of extra code to maintain, build, and test taxes the entire development effort. Dropping backward compatibility allows for deleting old obsolete software and tests and makes the entire development effort run faster and more efficiently.

Also, the extra cost in maintaining old interfaces and behavior as new behavior is created discourages the creation of or refactoring to new interfaces and behaviors. Instead of changing existing interfaces (which would break backward compatibility), developers are drawn to keep the same interfaces in place and then put in "work-arounds" to add new behavior. Often these tweaks are not done in a clean way and the resulting software loses consistent structure. The tendency for software to lose its internal structure under subsequent releases and modifications is known as *software entropy* in [5]. Without regular refactoring to clean up the interfaces and design of the software (which often breaks backward compatibility), such software dies a slow death of rising "technical debt" [12]. However, when backward compatibility is not maintained, then the software can be freely refactored to maintain "conceptual integrity" [5].

Therefore, while it is easy to see the critical need for maintaining some backward compatibility as described in Section 6.2, we have also acknowledged that maintaining backward compatibility over a long-lived piece of software can impart significant extra cost or significantly increase the software entropy and technical debt of the software that will eventually make the software unable to continue to be changed from a cost/benefit point of view. In other words, the code will become unsustainable (i.e., not self-sustaining) software.

6.4 Regulated Backward Compatibility: Defined

In order to balance the need and benefits of maintaining backward compatibility described in Section 6.2 against the costs of maintaining backward compatibility described in Section 6.3, we here define a compromised approach called *Regulated Backward Compatibility*. In this approach, sufficient windows of backward compatibility are maintained in order to achieve most of the benefits of maintaining backward compatibility but backward compatibility can be dropped periodically to avoid the accumulation of technical debt and extra testing related to maintaining backward compatibility.

The TriBITS approach for implementing regulated backward compatibility is inherent in the TriBITS version numbering scheme **X.Y.Z**[3] where

- **X** defines a backward compatible set of releases,
- **Y** is the major release (taken off of the master branch) in backward compatible set X, and
- **Z** is the minor release off the release branch X.Y.

The numbers Y and Z use even numbers for releases and odd numbers for development versions between releases The release number X.Y.Z is given in integer form in macros in the configured header file `<PROJECT>_version.h` (e.g. `Trilinos_version.h`). This numbering scheme X.Y.Z with even/odd numbers allows client code to manage different versions of the TriBITS project just by using pre-processor logic.

Side Node: A special exception to the even/odd numbering method for release/development versions is made when moving from X to X+1 (e.g., 11.6 to 12.0 releases). Once the last major release in the set X.Y is branched (e.g., 11.6), then the version of the development sources are immediately changed to X+1.0 (e.g., 12.0) instead of X.Y+1 or X+1.-1. This convention allows downstream customer codes to use ifdef logic based the unsigned integer version number in the file `<PROJECT>_version.h` (e.g., `Trilinos_version.h`) to show the major switch in backward compatibility sets for those that are building against the development version of Trilinos, yet does not require support for negative version numbers.

To demonstrate how the backward compatibility is managed, consider the release time-line with backward compatibility indicators shown in Figure 3. This scenario shows four major releases in the backward compatible set 11.Y (11.0, 11.2, 11.4, and 11.6). In the set 11.Y, each version is compatible with all of the prior versions. For example, 11.6 is compatible with all of the previous versions 11.0, 11.2, and 11.4. However, the transition from 11.6 to 12.0 drops guarantees of backward compatibility between the two versions. Section 6.5 describes how this transition to the next backward compatible set is handled.

With this approach, the trade-off between preserving backward compatibility verses controlling the cost of backward compatibility is managed by how many releases X.Y are in a backward compatible set X. Let N be the number of releases X.Y in the backward compatible set X. If N is small, then less effort is expended on maintaining backward compatibility but there may be more problems with customer upgrades. For example, if Trilinos only had two major releases X.0 and X.2 in a backward compatible set X, it would be difficult to accommodate a usage scenario with

[3]The version numbering scheme X.Y.Z is optional when using the TriBITS build system.

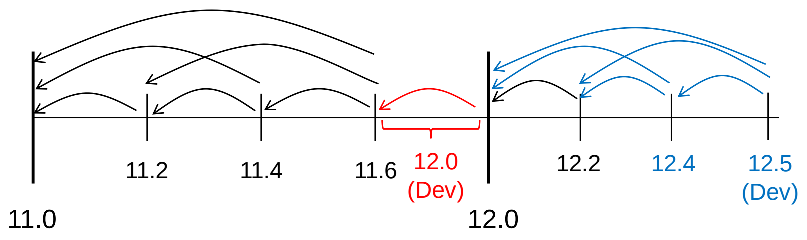


Figure 3. Regulated backward compatibility release time-line.

three dependent customer application codes as shown in Figure 2. Alternatively, if N is large then backward compatibility is maintained for long periods of time over many releases. This enables a greater number of usage and integration scenarios for downstream customer codes but the cost of maintaining backward compatibility may become very large.

6.5 Regulated Backward Compatibility: Details

There are two critical aspects related to the implementation of regulated backward compatibility that must be discussed: backward compatibility testing, and the process of breaking backward compatibility.

Within a backward compatible set X, it is not enough to simply try or desire to maintain backward compatibility, it must be tested. Testing must be performed to ensure that each subsequent version X.Y is backward compatible (to some level) with the previous versions X.Y-1, X.Y-2, etc. Testing of backward compatibility is accomplished in Trilinos by building a suite of user-oriented tests for an older version of Trilinos (e.g., 12.0 or 12.2) against the installed headers and libraries of a newer version of Trilinos (the current release 12.4 and development version 12.5 in Figure 3). Note that backward compatibility is only actively tested for the current release (e.g., 12.4) and the current development version (e.g., 12.5) since the older releases should all be static. For example, in the scenario shown in Figure 3, Trilinos release 12.4 would be tested against 12.2 and 12.0 test suites and Trilinos development version 12.5 would be tested against 12.4, 12.2, and 12.0 test suites. Therefore, the amount of backward compatibility testing needed scales just linearly with the number of releases in a backward compatibility set X. If N is the number of releases in set X, then $2N - 1$ backward compatibility builds must be performed on a regular (nightly) basis. While not perfect, this approach provides developers and users some assurance that backward compatibility is being maintained.

Note that by using this testing approach, the level of confidence in the maintenance of backward compatibility is only as good as the quality of the user-oriented tests. This test suite should likely contain the entire verification and acceptance test suites if possible. The unit test suite could also be included in the set of tests but that might overly constrain backward compatibility, but on the other hand would strengthen the testing of backward compatibility.

Note that the number of backward compatible releases in the set X determines how many nested collections of downstream software can be kept integrated, assuming they use the Punctuated Releases approach to software integration [2] where each application software just builds against a static release of the upstream package and other external software.

For example, if Trilinos provides backward compatibility for three contiguous releases, it allows for up to three nested client application codes to depend on each other and Trilinos, as shown in Figure 4. The releases of the three applications are depicted in Figure 5. Here new releases for App1 K+1, App2 M+1, and App3 J+1 are put out against three consecutive releases of Trilinos such that the final release App3 J+1 (against App2 M+1, App1 K+1 and Trilinos X+3) has the most recent versions of Trilinos, and each of its upstream dependent Apps. Therefore, if there are only N consecutive releases of Trilinos, then at most only N sets of nested applications can be supported in the most general case (i.e., where each App upgrades to the most recent Trilinos release currently available). And this can be done without any coordination between the Trilinos and the downstream development teams. If backward compatibility is not maintained for enough consecutive releases, then some amount of coordination and negation are needed between Trilinos and the various App development teams. This type of coordination is neither desirable nor practical in many cases.

The second and more challenging aspect to implementing regulated backward compatibility is handling the dropping of backward compatibility between backward compatible sets X and X+1 while easing the transition for downstream users and development teams. Simply putting out a release that breaks backward compatibility and breaks user code without any warning or means to help them upgrade their code is counterproductive. Even worse is the situation where changes in backward compatibility allow user code to build but the code then fails in subtle ways at runtime.

So how is the transition between backward compatible sets (e.g., from 11.6 to 12.0) managed? Here are some guidelines for managing the breaking of backward compatibility when transitioning between sets X and X+1:

- *Prepare users for the break in backward compatibility*: Try to to provide some preparation for breaking backward compatibility in previous releases. The best way to do this is to mark code as "deprecated". The GCC and Intel C++ compilers support a `__deprecated__` property that can provide compile-time deprecated warnings for classes and functions. There are many strategies for appropriately deprecating code to allow users a means to perform smooth transitions to new functionality before backward compatibility is dropped.
- *Fail big and hard when backward compatibility is dropped*: When breaking backward compatibility, client code yet to be upgraded should break in an obvious way. Ideally, non-upgraded client code should not even compile and the compile errors should be as obvious as possible. If compile errors are not possible, link-time errors would be the next best alternative. If compile or link errors are not possible, then run-time errors should be generated with good error messages helping the user to know what needs to be changed. Simply letting non-upgraded client code compile, run, and fail at runtime in non-obvious ways should be avoided if at all possible.
- *Provide for safe straightforward upgrades*: When breaking backward compatibility, try to provide the user an easy and error-proof approach to upgrade their code. For example, when changing the name of a set of classes and functions, provide a sed-like script that the user can run on their code to preform the name changes. This can be difficult for raw class and function names. However, for globally unique identifiers (like C function and macro names) these sed-like scripts are very safe.

Here are some guidelines for deprecating functionality to prepare users and downstream

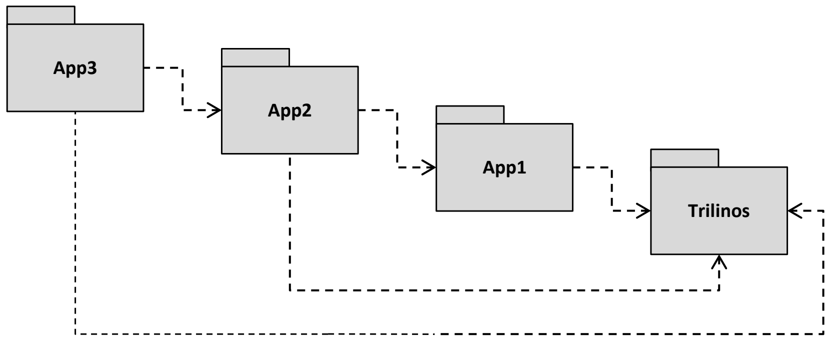


Figure 4. An example of a chain of three applications that depend on Trilinos and each other linearly.

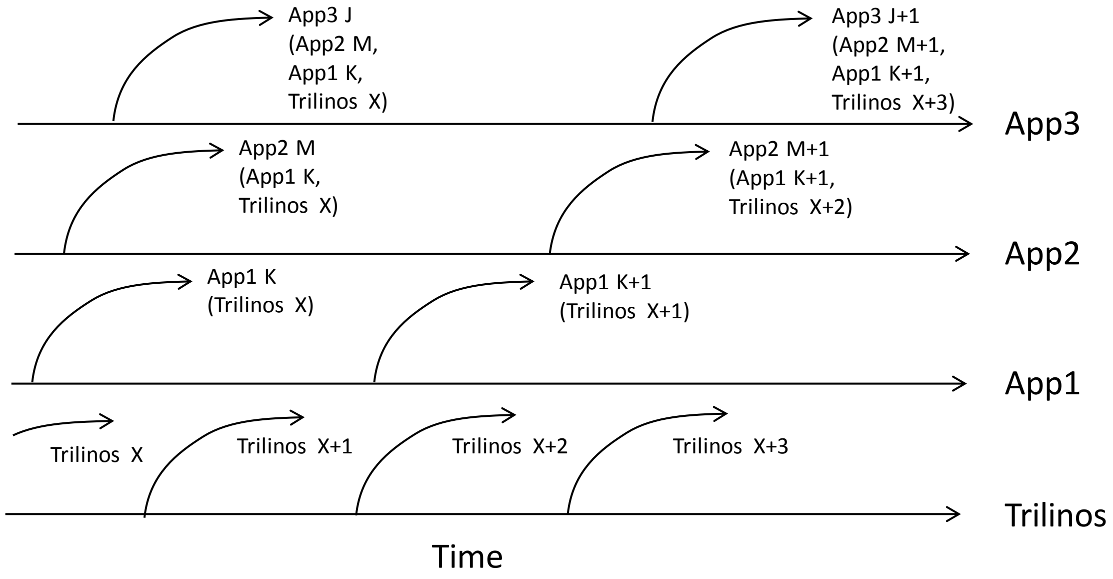

App3 J (App2 M, App1 K, Trilinos X)
App3 J+1 (App2 M+1, App1 K+1, Trilinos X+3)
App3
App2 M (App1 K, Trilinos X)
App2 M+1 (App1 K+1, Trilinos X+2)
App2
App1 K (Trilinos X)
App1 K+1 (Trilinos X+1)
App1
Trilinos X
Trilinos X+1
Trilinos X+2
Trilinos X+3
Trilinos
Time

Figure 5. An example of the releases of chain of three applications that depend on Trilinos and each other linearly.

developers to upgrade their code:

- *Deprecate classes and functions using the standard `<UCPACAKGENAME>_DEPRECATED` macro*: This macro expands to the GCC `__deprecated__` attribute when the TriBITS project is configured with `<PROJECT>_SHOW_DEPRECATED_WARNINGS = ON`.
- *Deprecate macros by calling dummy deprecated functions using the standard `<UCPACAKGENAME>_DEPRECATED` macro*: Any macro that can work by creating a nested scope as { `SomeDeprecatedFunction(); ...` } can use this approach. Other macros that can't create a nested scope to call a deprecated function like this are more problematic.
- *Deprecate old class names using deprecated typedefs or macro defines*: When changing the names of classes, keep the old names using safe deprecated typedefs for non-templated classes. However, for templated classes, C++98 does not allow the use of typedefs and therefore macro defines must be used to keep the old class names in a way that works with templates. These macros, however, cannot be deprecated (but the old deprecated header file they are in can be, see below).
- *Deprecate header files by inserting protected `#warning` directives*: A header file would be deprecated, for example, when changing the name of a class which would therefore require changing the name of its header and source files.

Examples of the usage of the above deprecation approaches can be found throughout Trilinos. By carefully deprecating code, downstream users and developers can safely and incrementally refactor their code to remove use of deprecated features. This refactoring to upgrade code be done in a relaxed and safe way since the changes can usually be made in small batches which include rerunning their test suites. Over time, downstream code developers can remove all deprecated warnings which will allow them to build and run successfully after backward compatibility is dropped.

In the ideal case, the transition from X.Y to X+1.0 (e.g., 11.Y to 12.0) would be accomplished by simply having the downstream code build against the last compatible release in the set X.Y (e.g., 11.6), remove all of the deprecated warnings (either manually or through provided sed-like scripts), and then transition to X+1.0 with no further changes. While this is the safest and most attractive approach for dealing with backward compatibility for downstream users and developers, there are some types of changes that are very hard to manage in this way. For example, if it is desired to change the names of a large number of templated C++ classes, it may be more desirable to instead wait until the last release in X is put out (e.g., 11.6) and then create a sed-like script to change a bunch of class names and file names all at once. This sed script would be run on the project code base right away and on the all downstream software projects that do Almost Continuous Integration [2] with the TriBITS project. This would also work well for downstream projects that do Punctuated Upgrades [2] for releases. As soon as they transitioned to Trilinos 12.0, for instance, they would run the provided sed-like upgrade scripts and be done (except for perhaps a little manual cleanup). However, downstream customer codes that do Release and Dev Daily Integration [2] would have a harder time because their code base could not be simultaneously compatible with both 11.6 and 12.0. In this case, they may need to drop building against either 11.6 or the development version leading to 12.0 (their choice will depend on their particular needs and constraints).

7 Detailed Discussion of the Proposed Lifecycle Stages

Now that the four different lifecycle phases have been introduced and the concepts of self-sufficient software and regulated backward compatibility have been described, the four different lifecycle phases are discussed in more detail.

7.1 Exploratory

As mentioned in Section 4, the Exploratory code stage is a place-holder for any research software that has not been developed in a Lean/Agile consistent way and is not an appropriate direct foundation for later development of production-quality software. For example, software with very low coverage of testing can likely not be considered to be Research Stable software. However, such software will be written for rapid prototyping purposes.

The Exploratory code phase and lax practices associated with it should only be used for software and algorithms that are not intended for product release. On the other hand, if the approaches or algorithms have a high probability of being successful, and for later becoming products, it will be more productive and more efficient to implement the software, from the very beginning, as Research Stable software as described below. Otherwise, if the algorithms and approaches in Exploratory software prove to be attractive for becoming products, new software should be written from scratch using the practices described below for Research Stable software.

7.2 Research Stable

The Research Stable stage is the initial stage for Lean/Agile consistent research software development. The goal in this stage is to experiment with various approaches, generate credible publishable results, and form the foundation for the probable future productization of the software.

All code in this stage should be written using Test-Driven Development (TDD), which will yield nearly 100% line coverage with minimal extra effort. This is depicted in Figure 1 with "Unit and Verification Testing" starting at relatively high level and staying high. This level of testing might seem excessive to some not accustomed to the TDD software development process, but even from a research perspective, these tests can help to avoid defects that would otherwise taint research results. (However, see the side-note on Page 20 for a discussion of lower-cost quick-and-dirty initial "unit tests".) Another important Lean/Agile development practice is to keep the code base very clean and clear, avoiding unnecessary complexity. Once good design and refactoring skills have been acquired, maintaining a clean code base is easier than crafting a messy/complex code. This is also depicted in Figure 1 by having "Code and Design Clarity" start out high and stay fairly high. Having a clean code base is also critical for research software, in order to verify that the algorithms and approaches described in a research publication are indeed what are implemented in the code. Also, as more journals start to require the submission and review of source code and test inputs used to generate published results, strong unit and verification tests, and a clean code base allow for a much easier review and greater chance of having a paper accepted.

In addition to being important from just a verifiable reproducible research perspective, reasonable unit and verification tests and a clean and clear code base form a critical foundation for creating

Side Note: *Levels of "unit testing" and lifecycle stage considerations:* When considering unit testing, we do not have to accept an all or nothing proposition. Minimally, unit tests must a) build fast, b) run fast, and c) isolate errors [3, 7]. This is contrasted with most system-level tests that do not build or run fast, and certainly do not isolate errors. However, it is likely that many developers who are new to test-driven development (TDD) and unit testing are scared away from the practice when they read some of the literature on TDD that shows lots of very fine-grained unit tests (sometimes dozens of unit tests for a single function). If a piece of software is important enough, then that level of unit testing (and more) will be necessary but it is more often the case that the initial "unit tests" might justifiably be fewer and more coarse-grained. In many cases, some coarser-grained tests can be written that lie between rigorous fine-grained unit tests and extremely course-grained system-level tests. For example, one of these medium-level tests might instantiate several objects of different types and call several functions before checking a final result. While these types of tests may not be the best at isolating errors, in many cases they can still otherwise build fast and run fast. If well crafted, these types of quick-and-dirty (using the term lightly) "unit tests" can often achieve high coverage and often do not take much longer to write than a system-level test or the labor to manually run and verify a result (say by running the code in a debugger and stepping through it to see values). After these quick-and-dirty unit tests are in place, as a piece of software matures and is more heavily used, then finer-grained unit tests can be written to define boundary cases and to otherwise better define and protect the software. Therefore, using TDD with quick-and-dirty coarser-grained unit tests need not take any longer than the more typical manual validation-based development process. These automated tests then protect the software going forward for all time (which is not true of manual validation-based testing). In addition, having a good unit test harness for your language and programming environment can also greatly reduce the overhead of doing TDD and unit testing. Making an investment in an easy-to-use unit test harness pays off many times over in later usage.

self-sustaining software (see Section 5) and providing a foundation for possible later productization of the code.

While Lean/Agile Research Stable software needs to have high-quality unit & verification testing and clean code, it need not have much (if any) documentation, it can have very little error checking of invalid user input and no feedback in case of incorrect input (it can just assert() and die), and it may not have any acceptance testing (because there are no real customers). While addressing these issues is critical for high-quality production software, they are of little use for pure research software (that is used by only the core developers or very friendly users). In addition, Research Stable software need not maintain any backward compatibility at all if there are no down-stream customers (and there should not be too many down-stream customers for a pure research code). Any downstream customers that would be affected by a break in backward compatibility should be readily accessible to upgrade when non-backward compatible changes are made.

Finally, Research Stable software really only needs to build and run successfully on the primary development platform (currently GCC 4.5+ for Trilinos). While this might sound like a bad way to develop software that is being targeted for possible later productization, consider that the TriBITS system automatically inserts strong compiler warning options into every build (see Section 3), even for Exploratory and Research Stable code. Extensive experience has shown that C++ code that builds clean of warnings with these strong compiler warning flags and has a good test suite (which Research Stable software does by definition) typically has few serious problems when porting to other platforms. Therefore, portability to other platforms is not considered a concern for Research Stable software in order to enter the next phase, the Production Growth phase.

The last issue to consider in the lifecycle of software is space/time performance. For most research-based software, algorithmic properties are the primary concern, not raw performance. In addition, premature code optimization efforts can greatly damage the clean design and layout of the code. Therefore, code optimization to improve space/time performance metrics should only be considered in the Research Stable phase when the focus of the research is on low-level performance. Otherwise, the focus should be on writing clean well tested software.

If the approaches or algorithms being researched have a low probability of being successful, being submitted for publication, or being considered for later productization, then it may be more efficient and more appropriate to develop the software as Exploratory software as described above.

7.3 Production Growth

After the algorithms in a piece of software have been sufficiently proven in the Research Stable phase, the software is a candidate for productization. Since the Research Stable code already has good unit and verification testing and has a clean design and source code, it is immediately ready to enter the Production Growth phase.

When software enters the Production Growth phase, the purpose is to start adding those features needed to make the software more usable for downstream clients and add functionality needed to support their missions. However, while a primary focus is on starting to improve the production quality of the software, the software can still be used for new research to some extent but the overhead of doing research in the code base will become larger and larger as the software becomes

Side Note: *Refactoring Exploratory code to Research Stable Code:* It is also possible to take Exploratory software and turn it into Research Stable software that can seed the Production Growth phase but it would require a large refactoring effort driven by intense unit and verification test development. The large cost needed to get Exploratory code to the standard of quality needed for Research Stable code (the precondition for entering the Production Growth phase) is usually not undertaken and therefore software that is not developed from the beginning with strong testing with continuous refactoring to maintain clear design and code typically never achieves the properties necessary for self-sustaining software. Therefore, again, Exploratory software should likely be thrown away and rewritten from scratch as Research Stable software.

more productized (and has to maintain better backward compatibility, needs better documentation, etc.). Continued research can be conducted in related subpackages to minimize the impact on the body of software under consideration that is beginning productized.

In the Production Growth phase, the user-oriented elements start to get fleshed out, as depicted in Figure 1, including adding documentation and tutorials, adding more tests for boundary conditions for invalid user input (and improving error messages that are generated), and starting to become increasingly more careful about maintaining backward compatibility. All the while new features are developed and functionality changes, the unit and verification testing is continued and even expanded to cover more boundary cases and other scenarios. At the same time, the code is continually being refactored to keep the design simple and the code layout clean and understandable. This is done while maintaining backward compatibility by deprecating features, thereby giving downstream developers the chance to safely upgrade their code to keep up with the changes (see Section 6.5).

This is also the stage where more serious customers can start to depend on the software because they know that backward compatibility will be handled more carefully and the development team will be trying to improve documentation, error checking and reporting, and will be integrating acceptance tests that the customer develops with them. At the beginning of the Production Growth phase, backward compatibility may not be maintained very well and downstream developers may have a hard time keeping up with the changes. However, quickly the upstream team should start doing a better job of deprecating features and smoothing the transition for downstream customer code and developers. The relationship with downstream customers is greatly facilitated if Almost Continuous Integration [2] is used where deprecated features are seen right away and breaks in backward compatibility are caught and addressed quickly.

As more customers are added, more platforms will need to be supported beyond just the single primary development platform. Therefore, the Production Growth phase will see the portability of the software increased. Likewise, in order to entice new customers or to even make the software viable to other customers, space/time performance characteristics (including parallel scalability) will need to be improved. The big danger here is that overzealous code optimization efforts will destroy the clean and clear structure of the code, which if allowed to happen, would severely increase the technical debt of the software and risk its long term viability as self-sustaining software.

7.4 Production Maintenance

Once a piece of software has obtained a reasonable level of maturity and higher levels of quality in documentation, user input checking and feedback, acceptance testing, maintaining backward compatibility, portability, and performance, then the software is a candidate to move into the final Production Maintenance code phase. Once software enters this phase, it should likely not be developed quite as actively and there should be less need to change and deprecate interfaces. Also, since the internal structure of the software will still be refactored to maintain a clean design and structure, the design and code clarity will stay high as the software is maintained.

In the Production Maintenance phase, a wider range of customers might be attracted to the code because if its greater stability (i.e., not needing to deal with a lot of deprecated features and the subsequent refactorings to absorb breaks in backward compatibility). Therefore, significant growth in the development of acceptance tests for new customers is likely to continue. However, we would expect that the functioning and the purpose of the software will have been developed to the point where new functionality is handled by writing new code to specialize behavior and not changing existing code (e.g., the Open-Closed Principle (OCP) [8]).

By the time software enters the Production Maintenance phase, it should have very good sustained portability. This portability will be maintained through automated nightly testing and will therefore not regress.

Software in the Production Maintenance phase may need to drop back down to the Production Growth phase if a significant change in functionality is needed, thereby requiring a major refactoring effort that might require significant changes in code behavior and changes in backward compatibility.

7.5 End of life?

Once a piece of software achieves the Production Maintenance maturity level, then it is a well tested, well documented, robust piece of software which (hopefully) has a large customer base. Since the software satisfies all of the criteria of self-sustaining software (and all of its dependencies are also self-sustaining software by definition), then the software will likely live on for many years or even decades to come, being ported to new architectures and taking on new functionality as needed. When the time comes for the original developing team or organization to discontinue development and support of the software (which will eventually happen for every piece of software), then the software will still live on if there are downstream customer codes that still use it. These customer organizations can band together (or not) and continue to perform the most basic maintenance to support new platforms or make minor changes as needed. These organizations can be confident that they can perform these tasks even though they were not the original developers because a) the software has a clean design and has clear understandable code, and b) the software is exceptionally well tested with both unit and verification tests and also with good acceptance tests that they have helped to develop. In other words, the software is self-sustaining!

On the other hand, if technology changes significantly, or if the implementing programming language falls completely out of style and support, or if customers move on to other approaches or software packages, then the software in question can stop being built and tested on a regular basis (and thereby go quietly into the night). However, given access to compilers and compatible computing platforms, the software can still come back to life and can continue to be maintained because it is clean and clear, is exceptionally well tested, has good documentation, and is robust to user input errors.

8 Software Usefulness Maturity and Lifecycle Phases

Now that we have defined the TriBITS Software Lifecycle phases, it is important to note that maturity in a software engineering sense is not necessarily correlated with maturity in the usefulness of a software package. A package can theoretically be in any of the TriBITS phases but its true usefulness not be indicated. Furthermore, as is clear from the collection of useful CSE software, a package may be very useful but not follow any prescribed software lifecycle and have minimal testing coverage (and therefore is very fragile under refactoring activities).

Even so, practically speaking, the usefulness of a software package is typically the driving force behind moving it from one phase to another in the TriBITS Lifecycle, and provides the incentive for investing in software engineering activities to preserve the package's future usefulness. Furthermore, funding for a software product is often directly connected to its perceived usefulness. Therefore, any package that is in the Production Growth or Production Maintenance phase will almost surely be very mature in its usefulness.

Unfortunately, the opposite is not necessarily true. A package that is very mature in its usefulness is not necessarily very mature in a software engineering sense. In fact, paradoxically, unless a software development team has carefully managed the engineering of a useful software package, there is a good chance that the package has, or will eventually, become practically unchangeable. In this situation, the software has matured and is very useful, but has outgrown its testing coverage, has very complicated internal logic and has rigid interface constraints (because it is extremely valuable to its user base and change management has not been engineered into the package). In this scenario the package development team has a difficult choice to make: It must either address the deficiencies in the mature package or start a new package, which are both very expensive. The former choice is expensive because it disrupts the activities of the existing user base and is intrinsically distasteful and difficult for developers. The latter choice is expensive because usefulness maturity happens over a long period of time and requires extensive interaction with the user base.

An assessment of usefulness maturity is something that users expect when they consider adopting a software package. However, it is not something the TriBITS Software Lifecycle model addresses. Instead we highlight that, since the intention of the CSE software community is to develop useful software that will retain its value and be adaptable over a long period of time, investment in the software engineering processes and practices advocated by the TriBITS model is very important. Furthermore, as we discuss in Section 11 below, it is never too late to start using rigorous software engineering processes on an existing, highly useful piece of software. In fact, doing so can extend the life of such a product indefinitely.

9 Risk Analysis and Acceptance Testing

Risk analysis is a critical and sometimes overlooked aspect of a complete software lifecycle model. Here, we mostly consider the risk analysis of CSE software that provides foundational algorithms and infrastructure but may not provide end-user application software. Trilinos is an example of such a collection of CSE service software without direct CSE simulation end-users. However, many of the issue described here also apply the end-user application software as well.

Two important aspects of CSE services software (e.g. Trilinos) that factor into the risk analysis process are that CSE services software itself does not contain any end user CSE applications and the requirements that go into services software development flow down from developers of end-user applications or research codes. Therefore, the participation of customer codes is critical in the risk analysis process.

Most risks cannot be eliminated, but analyzing and mitigating identified risks can significantly increase a project's chances of success. The following risks are applicable to most CSE software projects:

- Developer Turnover
- Funding Gaps
- Third-Party Software Dependencies
- Programming Language Selection
- Algorithm Selection and Implementation
- Verification and Validation

Producing self-sustaining software is a key mitigating factor when considering these and other important CSE software risks. A closer look at each risk will further illustrate the importance of self-sustaining software in mitigating the risk.

Developer Turnover: Many software projects have one or more developers whose deep knowledge of the software seems indispensable and absolutely critical to the future of the project. Clearly, this situation represents an enormous risk. While losing such a developer will no doubt adversely affect a project, a clean design, clearly written code, and sufficient documentation will allow another developer to more easily assume the development tasks of the departing individual. Furthermore, a well-designed, complete testing infrastructure, and sufficient resources for exercising the suite of tests properly will allow remaining developers to more effectively maintain the code base after an individual with critical knowledge has left the project.

Funding Gaps: Discontinued, decreased or inconsistent funding is a risk that is similar in some ways to losing a key developer. The risk mitigation strategy is also similar. In this case, a clean design, clearly written code, sufficient documentation, and open source licensing would allow any able person, perhaps a user, to assume maintenance and development responsibilities. Production maintenance phase software, which requires excellent checking for user input errors and handles those errors gracefully, and provides extensive documentation is positioned most firmly to remain heavily used in the case where funding for the project becomes an issue.

A well-designed, complete testing infrastructure is again critical. In this case it is particularly important that the testing system be easily deploy-able on other systems. On the application end, a robust set of acceptance tests for each user code becomes even more critical. Developers should assist key users in setting up continuous integration processes as part of mitigating this risk.

Another way to mitigate the risk of funding problems is to seek a diverse stream of funding, rather than relying on a single, large funding source, even in the case where such a source is available. Also, self-sustaining software is cheaper to maintain in the long-run, increasing the chance that a smaller funding source may be able to meaningfully sustain the effort at some level.

Third-Party Software Dependencies: When assessing third-party software for possible use within a code base, consider whether or not the the third-party software is itself self-sustaining. If not, strongly consider either not using the software, making the dependence on the software optional, or have a tentative plan in place in case the dependency becomes problematic. A tentative plan might include removing the dependence, replacing the dependence with similar third-party or in-house software, or (if allowed by the third-party license) maintaining a custom version of the third-party software that is released along with the primary project code.

In the case where the third-party code is generally reliable and self-sustaining, no action is required but the possibility that the code could deteriorate in quality or cease to be supported should be considered, in which case one of the above actions would need to be pursued.

Programming Language Selection: We encourage teams to select only widely available and maintained languages, except in the case where the purpose of the project is specifically tied to another language. A language with a large body of software that depends on it is not likely to disappear. This risk should be addressed early in the design phase prior to the implementation phase. Be sure to consider platforms and compilers that are not currently supported, but may need to be supported in the future.

Algorithm Selection and Implementation: Are the limitations of the algorithms and the approaches taken in the software too severe (in terms of speed, robustness, or otherwise) to be of real use to anyone to bother with worrying about trying to productize the software? Again, mitigation of this risk starts in the design phase. Mitigation also requires early user feedback. When conducting algorithmic research, there is always the risk that the resulting algorithm may not meet the needs of the customer; concentrate on mitigating the risk that a poor design or lack of communication leads to the failure of the approach, rather than on the risk inherent in research.

Verification and Validation: The importance of Verification and Validation (V&V), and in particular the risks associated with relying heavily on validation while under-emphasizing verification, were discussed in Section 2. A validation-centric approach does not result in self-sustaining software.

Considering the example of the Trilinos project, Trilinos developers are responsible for fundamental code verification and much of the structure needed for doing solution verification [15]. The end-user application developers shoulder the primary responsibility for performing validation, especially at scale.

The verification risks can be mitigated with a strong testing infrastructure as described above. Preferably with input from end users, some validation may be incorporated into the testing infrastructure also.

Again, end user applications play a critical role in risk mitigation, especially when it comes to validation. In CSE, it can be argued that the best acceptance test suite for a given piece of upstream numerical library or other intermediate CSE software is the end-user application's own verification test suite [2]. It can be further argued that the lowest risk is achieved by having the end-user customer application code teams perform Almost Continuous Integration [2] (as is currently performed for key Trilinos customers such as SIERRA, CASL, and several other down-stream software projects). In addition to frequently running the end-user application's own verification test suite, the end-user application developers can also work with the Trilinos package developers to create stand-alone acceptance tests that can be collected with the Trilinos package's own test suite apart from the end-user application code. These stand-alone acceptance tests can then be run as part of the regular automated testing as the software is maintained [6]. Such a model naturally keeps open lines of communication between a code team and its customers and decreases the chance that the code team will fail to address its customers' needs.

10 Comparison to a Typical CSE Lifecycle Model

A typical non-Lean/Agile software development process used to develop CSE software (as determined by personal experience and a number of studies) would suggest that its quality metrics are like those shown in Figure 6, where unit and verification testing and code/design clarity are low and only get worse under maintenance. The reduction in unit testing typically occurs because as the software grows without maintaining a good architecture, the entangling dependencies make it more difficult to get objects into a test harness and the developers invariably fall back on system level acceptance (or regression) tests. At the same time, as the software is modified as new functionality is added without being refactored, the software becomes more convoluted and fragile and eventually dies the slow death of software entropy [5].

Note that the time-lines for user-oriented metrics for documentation, acceptance testing, user input validation, backward compatibility, portability, and performance would typically look more similar to a Lean/Agile method as show in Figure 1. This is because these user-oriented metrics are most directly seen by customers and (with the exception of space/time performance) often do not require significant refactorings or code changes that would require clean code or good testing to support. In fact, the lack of refactoring efforts in typical CSE projects might actually produce higher levels of backward compatibility than the proposed Lean/Agile lifecycle process being described here. What is not shown in Figure 1, however, is the added cost needed to get such a code to a higher quality state and maintain it as more features are added. Also, the lower the design/code clarity and unit & verification testing, the less likely a group outside of the original development team can maintain the software.

Depending on software such as this represents a large risk for downstream customer projects since this software is not self-sufficient and is unsustainable by any but the originating organization or team that created the software. The prevalence of this type of software in the CSE community is a major reason for the trepidation that many CSE groups have in taking on external software dependencies. Most of this apprehension is founded on real experiences of getting burned by bad upstream CSE software. The TriBITS lifecycle model is an attempt to reverse this trend (starting with Trilinos and then for related TriBITS-based projects).

11 Grandfathering of Existing Code and Unspecified Maturity

At the time of this writing, many of the established long-released CSE software products (e.g. Trilinos packages) don't meet the criteria for basic Research Stable code (not to mention Production Growth) because they lack a sufficient level of unit and verification tests or the design and/or internal structure of the software is too cluttered to be considered self-sustaining software. However, some of these established packages are highly useful (see Section 8) and have been hardened through extensive use by customers and bug fixes and they are not as actively developed and therefore, for certain common use-cases, are well validated (see Section 2). This is a typical way in which testing and hardening is performed in CSE and other domains but is not

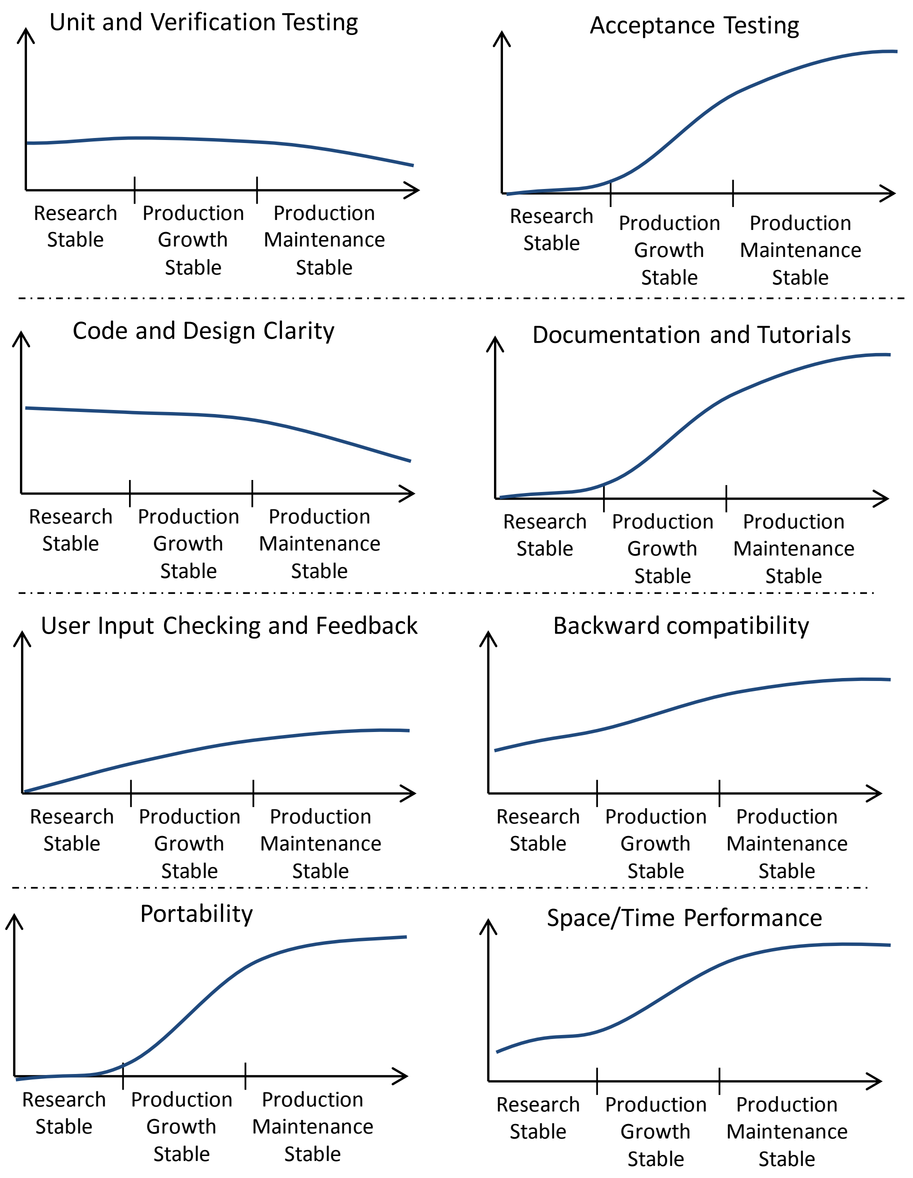


Figure 6. Example of the more typical variability in key quality metrics in a typical CSE software development process.

consistent with Lean/Agile and does not lead to self-sustaining software. While all of this is true, we will have to "forgive" this current generation of packages and grandfather them into the new lifecycle model. At the same time, this code is useful and represents important capabilities and needs to be maintained and improved. Therefore, this lifecycle model will allow these packages to be classified as Research Stable, Production Growth, and Production Maintenance packages as long as the package developers agree, going forward, to make all further modifications to the code using high-quality Lean/Agile consistent practices. The right way to maintain these important packages going forward is to apply the *Agile Legacy Software Change Algorithm* defined in [7] which states that every change to legacy code be carried out as follows:

1. *Identify Change Points*: Identify the code that needs to change, isolate its change points and sensing points.
2. *Break Dependencies*: Use hyper-sensitive minimal editing to break fundamental dependencies to allow the code being changed to be inserted into a unit test harness.
3. *Cover with Unit Tests*: Write new unit tests to protect the current functioning and behavior of the code that will be changed.
4. *Add New Functionality with TDD*: Write new, initially failing, unit tests to define the new desired behavior (or to reproduce a suspected bug) and then change the code incrementally to get the new unit tests to pass. This is the test-driven development (TDD) approach. All the time, be rebuilding and rerunning the newly defined unit tests to make sure changes don't break the existing behavior of the code.
5. *Refactor*: Clean up the code currently under test to reduce complexity, improve comprehensibility, remove duplication, and provide the foundation for further likely changes.

The above algorithm can be succinctly described as "cover", "change", and "refactor".

Any existing package that will be further changed and maintained according to the above-defined Agile Legacy Change algorithm can be classified as follows:

1. Grandfathered Research Stable Code
2. Grandfathered Production Growth Code
3. Grandfathered Production Maintenance Code

By applying the above Agile Legacy Code change algorithm over and over again in small chunks over a long period of time, even the worst legacy software (i.e. with no tests and messy code) can slowly be turned into quality software that will become easier to change. If grandfathered software is changed enough using the Agile Legacy Change process, it may eventually achieve a level of design and clarity and unit and verification testing that it can legitimately be considered to be Lean/Agile consistent software and the prefix "Grandfathered" can be dropped.

Given the powerful incremental refactoring and testing approaches described above, the message should be that even a piece of software that might otherwise be considered hopeless, in reality might in fact be relatively inexpensively resurrected and refactored into the next generation of self-sustaining software. Here, the claim of "relatively inexpensively" is compared to the total cost of writing new software from scratch to replace the existing legacy software which can be a huge cost; much more than most people think (see the discussion of "green field" projects in [7] and the Netscape 6.0 experience[4]).

[4] `http://www.joelonsoftware.com/articles/fog0000000069.html`

On the other hand, if there are package development teams of existing legacy software that are not committed to using the Agile Legacy Change process to further maintain their code (and therefore incrementally turn their software into self-sustaining software), then their packages cannot be considered to be "grandfathered" into this lifecycle model and therefore must be content to be categorized as Unspecified Maturity software. In this case, the hosting project will not comment in any official way at all on the quality or maturity of a package categorized as Unspecified Maturity. Package software classified as Unspecified Maturity code requires that prospective customers independently examine/test/evaluate the software itself and/or directly communicate with the given package development teams in order to determine quality in order to determine if they should use the package or not.

12 Summary and Next Steps

We need to change the way the CSE community writes and future develops research-based CSE software, including Trilinos software. Much of the current research software is in a state where there is not enough confidence in the validity of the results to even justify drawing conclusions in scholarly publications (there are some examples of where defective software gave wrong results and hurt the creation of knowledge in the research community [10]). CSE development teams should be using TDD and need to write some unit and verification tests, even if the only purpose of the software is to do research and publish results. We also need some type of review of the software to provide the basis for publishing results that come from the code (but of course the typical journal peer-review process will not do this).

A follow-on to this document might be another document that defines an official assessment process for packages in TriBITS-based projects (e.g., Trilinos) to assign their current maturity level. This assessment process would need to be performed on a regular basis through code reviews and the examination of various quality metrics. Doing this would mean defining some metrics that would then be applied in making these assessments. This sounds much like a CMMI assessment but this is something that needs to be considered. One might just have a table of various metrics of software quality and then maintain scores for the various packages. Some of these metrics could be filled in automatically (like code coverage) but most would be somewhat subjective and would need people to make reasonable determinations. The final determination of a package's (or subpackage's) phase / maturity level could be determined by the minimum score in a given category or by some other criteria.

Finally, as mentioned in Section 11, existing packages will necessarily need to be "grandfathered in". This newly defined TriBITS lifecycle process will not magically get everyone who develops software to develop their software in a Lean/Agile consistent way (i.e. with high unit and verification testing right from the very beginning with a clean code base). There is a large cultural issue that will need to be addressed and this document is just a step along the path to getting the CSE community (such as the Trilinos development community) to where it needs to be with respect to software quality in various projects.

References

[1] S.L. Bain. *Emergent design: the evolutionary nature of professional software development.* Net Objectives, 2008.

[2] R.A. Bartlett. Integration strategies for computational science. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 35 –42, 23-23 2009.

[3] K. Beck. *Test Driven Development.* Addison Wesley, 2003.

[4] K. Beck. *Extreme Programming (Second Edition).* Addison Wesley, 2005.

[5] F. Brooks. *The Mythical Man-Month (second edition).* Addison Wesley, 1995.

[6] E. Evans. *Domain-Driven Design.* Addison Wesley, 2004.

[7] M. Feathers. *Working Effectively with Legacy Code.* Addison Wesley, 2005.

[8] R. Martin. *Agile Software Development (Principles, Patterns, and Practices).* Prentice Hall, 2003.

[9] S. McConnell. *Code Complete: Second Edition.* Microsoft Press, 2004.

[10] G. Miller. A scientist's nightmare: Software problem leads to five retractions. *Science*, 314(5807):1856–1857, 2006.

[11] R. Pawlowski, R. A. Bartlett, N. Belcourt, R. Hooper, and R. Schmidt. A theory manual for multi-physics code coupling in LIME. Sandia Technical Report SAND2011-2195, Sandia National Laboratories, March 2011.

[12] M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development.* Addison Wesley, 2007.

[13] D. Post and L. Votta. Computational science demands and new paradigm. *Physics Today*, 58(1):35–41, 2005.

[14] K. Schwaber and M. Beedle. *Agile Software Development with Scrum.* Prentice Hall, 2002.

[15] T. Trucano, D. Post, M. Pilch, and W. Oberkampf. Software engineering intersections with verification and validation (v&v) of high performance computational science software: Some observations. Technical report SAND2005-3662P, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2005.

[16] M. VanDerVanter, D.E. Post, and M.E. Zosel. Hpc needs a tool strategy. Technical report LA-UR-05-1592, Las Alamos Laboratories, 2005.

[17] James M. Willenbring, Michael A. Heroux, and Robert T. Heaphy. The Trilinos software lifecycle model. In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, page 186, Washington, DC, USA, 2007. IEEE Computer Society.

A Comparison to the Trilinos Lifecycle Model 1.0

Here we discuss the similarities and differences of the originally defined Trilinos lifecycle model [17] (which we will refer to here as the Trilinos Lifecycle 1.0 model) and the new TriBITS lifecycle model described in this document.

The Trilinos Lifecycle 1.0 model defined three basic phases:

- *Research* Phase:
  - The research proposal is the project plan.
  - Software is placed under configuration control as needed to prevent loss due to disaster.
  - Peer reviewed published paper(s) is primary V & V.
  - The focus of testing is a proof of correctness, not software.
  - Periodic status reports should be produced.
  - A lab notebook, project notebook, or equivalent is the primary artifact.
- *Production Growth* Phase:
  - Agile methods (with associated lifecycles) are encouraged.
  - All essential ASC SQE practices performed at an appropriate level (predetermined during promotion event from the research phase).
  - Artifacts should naturally "fall out" from SQE practices and periodic status reviews and management reports.
  - Process improvement and metrics are appropriate.
- *Production Maintenance* Phase:
  - Software has stable requirements.
  - Most development is isolated to minor enhancements and bug fixes.
  - Better design documentation and internal documentation are developed for the software.
  - The software can be handed over to a different maintenance team.

The Trilinos Lifecycle 1.0 model also described specific promotional events that require tasks such as risk analysis, gap analysis, and explicit promotional decisions written up as semi-formal reports. There is also mention of getting the team to seek more training and the development of new processes. The document [17] mentions some set of required practices and processes and supporting artifacts before a package has been released (or get a waiver from the Trilinos Project Leader). It is said that it is only after sufficient artifacts have been approved that a package is eligible to go out in a Trilinos release. Also, the Trilinos Lifecycle 1.0 model document says that once a package is no longer supported that it will be removed from later Trilinos releases.

The biggest problem with the above described Trilinos Lifecycle 1.0 model defined in [17] is that it is a theoretical model. The model is only theoretical because, as of this writing, no Trilinos package has ever followed the outlined process of performing documented risk analysis and gap analysis, and then producing artifacts that are sent to the Trilinos framework list for review and archiving. No package has ever met the criteria necessary for the Production Maintenance phase and no package has ever been officially turned over to a "maintenance support team." Packages have only been abandoned by their primary developers without any plan for future support and (for the most part) have still been allowed to remain in subsequent Trilinos releases (with very few exceptions). There are no modern SE lifecycle processes that advocate this type of approach to developing software. There is likely not even a single Trilinos package that meets the criteria for the Production Growth phase in [17] which includes the specification for the use of Agile practices and the “essential ASC SQE practices” (which are quite detailed) and there is no documentation for what essential ASC SQE practices are followed and which are not and why.

A primary substantive problem with the Trilinos lifecycle 1.0 model is that it does not assume any quality software development practices are being used while the software is first being developed in the Research phase. Mention of Agile (or any other) quality practices does not come in until the Production Growth phase. By that time, there may be a significant amount of software that is well below standard in terms of design/code clarity and unit and verification testing. If this software is used as the direct seed for the Production Growth phase (which is what often happens), then there is typically such a large technical debt that it never gets addressed once more focused efforts towards productization begin. This is depicted in Figure 6 by the design/code clarity and unit & verification testing steadily going down as the software is further developed. The reason for this is discussed in Section 10. Software that begins life with this level of (absent) standards is equivalent to the Exploratory code category described in Section 4 and Section 7.1. As described in Section 7.2, it is possible for this type of software to be refactored into a form that is appropriate for production but that almost never happens in practice. People and funding agencies are not willing to pay down the technical debt and there is just too much inertia with the current design and development approaches and these plague the software for the rest of its existence in challenged attempts to turn it into production software. However, if the software developed in the Research phase is completely scrapped, it can be duplicated using Agile development practices as described for Research Stable code in Section 7.2. Therefore, the really only hope for the Trilinos Lifecycle 1.0 model is to completely scrap software developed in the Research phase and rewrite it from scratch in order to provide the seed for the Production Growth phase.

Another big difference between the Trilinos 1.0 and TriBITS lifecycle models is that the promotional event to the Production Maintenance phase. In the original 1.0 lifecycle model, the development team is supposed to create a bunch of project artifacts after the fact and/or refactor (or rewrite) much of the software from scratch. This will likely never happen, no-one will likely ever pay for this type of effort, and the developers of the package will likely never do it. Major batch efforts left for the end of development are seldom ever actually done in practice (hence “small batch theory” in Lean [12]). Instead, in this updated TriBITS lifecycle model, preparing for maintenance mode is accomplished incrementally by using Continuous Refactoring to keep the design of the code simple and by creating and maintaining a higher-level document that describes the key concepts as a road map (examples of these types of high-level documents from the DDD book are good). Of course it is also facilitated by having very good unit and other verification tests. There is no need for the types of expensive end-of-development artifacts that are described in [17] if the software has the properties of self-sustaining software (see Section 5).

Related to the issue of artifacts described above, yet another difference between the original 1.0 and the new TriBITS lifecycle models is that in the TriBITS lifecycle model there are no distinct promotional events or large batches of work necessary to move a package from one lifecycle phase to the next like in the original 1.0 lifecycle model. Instead, once enough of the attributes (e.g., quality of documentation, error reporting, portability, etc.) of a given lifecycle phase for a given package are met, a decision is made by someone or some group of people that the package is in the next stage and is then labeled as such. The independent evidence inherent in the software will support or refute such a claim and users can assess such an assignment for themselves (more on this possible assessment process in Section 12 .

Another major problem with the Production Maintenance phase in the 1.0 lifecycle model is that there is an assumption that the software could be redesigned and largely rewritten in order to allow the software to endure long-term maintenance by a separate software support team. The problem with this assumption is the fact that you can't really radically improve a piece of software with a redesign if you need to maintain strong backward compatibility. Inherit design flaws in the user interface will never be fixable unless you deprecate problematic aspects of the old user interface and replace them in a better design.

Another issue that is handled very differently between the Trilinos 1.0 and TriBITS lifecycle models is the issue equating "release" with "production" which was implicit in the original Trilinos lifecycle 1.0 model. Instead, in the new TriBITS model, a package can be released (i.e. provided to a set of potential users that are not the original developers) while being in any state of development. However, making this work requires that packages be properly categorized so that users know what to (or not to) expect when taking on a dependency on the package. For example, if it is advertised that package X is in the Exploratory phase, potential customers know that they should not rely on or trust the software in any way. However, having access to such low-quality research software can still be beneficial in order to help improve it or use ideas from the software to create better quality software. On the other hand, if Package Y is categorized as Production Maintenance code then users know that they can reply on the proper functioning of the software and will also know that they will be able to accept future versions of the software free of serious regressions or changes in backward comparability. In the new TriBITS lifecycle model, releasing or not releasing software is not the issue. The issue is the proper categorization of the released software and expressing the state of the software appropriately to prospective users. That puts the power in the hands of the user and they can make up their own minds for how to deal with different packages.

Another inconsistency between the Trilinos 1.0 and TriBITS lifecycle models is that the Trilinos 1.0 model's assertion that each Trilinos package can have their own completely independent lifecycle model is incompatible with the TriBITS lifecycle model's suggestion of providing an official classification of packages. Without some official uniform standards that come from these maturity level descriptions, end users will be on their own trying to determine what software they should use and which they should wait until it becomes more mature.

Finally, the last major difference between the TriBITS and Trilinos 1.0 lifecycle models is that the TriBITS lifecycle model does not assume that there is any need for an official "maintenance support team". The goal of the TriBITS lifecycle model is the development of self-sustaining software that can be maintained by anyone as needed and even by end users (who then send patches back to the development team to have it incorporated into the master sources).

However, the updated Lean/Agile TriBITS lifecycle model described here does not totally invalidate the basic ideas in the original Trilinos lifecycle 1.0 model. It does require a stronger standard for fundamental testing and other basic Lean/Agile methods in research software than what is currently in place in a project like Trilinos (which currently really has no standards for testing, given that there are released Trilinos packages with no automated tests at all and several other packages have no tests in either an MPI or SERIAL build).

B Implementation of the Lifecycle Model in the TriBITS System

Here, we consider some details about how the TriBITS lifecycle model may be implemented in the TriBITS build, integrate, and test software system or how the TriBITS system can better support the TriBITS lifecycle model.

Assuming a TriBITS-based project could officially assign maturity levels based on well defined quality metrics, in order to be useful, packages and subpackges could be be explicitly tagged by their currently assessed lifecycle maturity level, i.e., Exploratory, Research Stable, Production Growth, or Production Maintenance. The main driver for rigorously tagging software in the different levels would be to separately generate quality metrics and higher levels of testing for coverage and memory checking, for example. If a project does not have an automated and maintainable means to select software with different maturity levels, it will not be possible to reliably gather code coverage and other quality metrics.

Officially tagging software in the TriBITS system would be facilitated by officially tagging packages and subpackges in the TriBITS CMake build system so that someone or something, for example the testing processes, could enable all Production Maintenance code with `<PROJECT>_ENABLE_MINIMUM_MATURITY_LEVEL = PRODUCTION_MAINTENANCE` and all software that does not achieve that level of maturity would be disabled. Actually, in order to allow new capabilities to be constantly developed in existing packages, the lifecycle phase should be assigned to subpackages within a package to allow software of lower maturity to coexist in the same package with more mature software. For example, a package labeled as Production Maintenance could have one subpackage at the level Production Maintenance but could also have another subpackage at the level Production Growth and another at Research Stable or even Exploratory maturity level. Therefore, when a TriBITS-based project is configured with `<PROJECT>_ENABLE_MINIMUM_MATURITY_LEVEL = PRODUCTION_MAINTENANCE`, only the one Production Maintenance subpackage would be enabled. However, when a TriBITS-based project was configured with `<PROJECT>_ENABLE_MINIMUM_MATURITY_LEVEL = RESEARCH_STABLE`, then the Production Maintenance, Production Growth, and the Research Stable subpackages would all be enabled. In this way, the testing system (or users) could explicitly filter software according to the required level of maturity before running coverage or other types of testing to generate quality metrics. Producing specific quality metrics (coverage, valgrind testing, etc.) for each maturity level would require adding just three new nightly builds for the different values of `<PROJECT>_ENABLE_MINIMUM_MATURITY_LEVEL` of `RESEARCH_STABLE`, `PRODUCTION_GROWTH`, and `PRODUCTION_MAINTENANCE` would these extra tests would only be needed on a single platform so would not significantly increase testing overhead.

It is questionable how useful an option like `<PROJECT>_ENABLE_MINIMUM_MATURITY_LEVEL` would be for end users but it could be made available for their usage if they desired. For example, a given customer project might consider the maturity level as a factor when deciding what packages to depend on and which are not yet ready for their usage.

DISTRIBUTION:

1 J. A. Turner
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6164

1	MS 1324	R. W. Leland, 1400
1	MS 0828	K. F. Alvin, 1220
1	MS 1322	S. S. Dosanjh, 1420
1	MS 1318	B. A. Hendrickson, 1440
1	MS 0380	D. E. Womble, 1540
1	MS 1322	J. B. Aidun, 1425
1	MS 1319	J. A. Ang, 1422
1	MS 1320	S. S. Collis, 1442
1	MS 1318	R. J. Hoekstra, 1426
1	MS 1323	S. A. Hutchinson, 1445
1	MS 1318	J. R. Stewart, 1441
1	MS 1321	R. M. Summers, 1444
1	MS 0380	M. W. Glass, 1545
1	MS 0899	RIM-Reports Management, 9532 (electronic copy)

v1.32

Sandia National Laboratories