Overview of the TriBITS Lifecycle Model

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

Roscoe A. Bartlett
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831
Email: `bartlettra@.ornl.gov`

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. Indeed this lifecycle process, if followed, will enable large-scale sustainable integration of many complex CSE software efforts across several institutions.

I. Introduction

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 [1]. 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 researchers 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.

Lean methods refer to methodologies taken from Lean Product Development that have been adapted to software development [2], and *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 [3], [4], [5]. Some of these core practices include continuous integration (CI), test-driven development (TDD), (continuous) design improvement, collective code ownership, and coding standards [3], [5].

In this 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). While the practices and processes described in this proposed lifecycle model have been demonstrated recently in smaller software efforts (e.g. parts of a few mostly newer Trilinos packages and related projects) by

the authors and some of their collaborators, this effort is an attempt to better define the lifecycle model so it can be tested on a larger scale across many projects and organizations.

The rest of this document is organized as follows. Section III describes the current state of software engineering supported by TriBITS. Section IV gives an overview of the new TriBITS lifecycle model phases/levels 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 V. The difference between and relationship of software engineering maturity and software usefulness maturity is the topic of Section VI. This is followed by a comparison of the new TriBITS lifecycle model to a typical lifecycle model used by a CSE project in Section VII. 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 VIII. The TriBITS lifecycle model is summarized and next steps are given in Section IX.

II. 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 (i.e. acceptance testing) 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 algebraic equation 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 often occurs as part of the development process, since the customer is using the product as it is being developed. Furthermore, investment in product-specific unit and integration tests is expensive and may seem unjustified if the software product's future is uncertain.

> **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 comprised of more than 50 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.

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 becomes 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 [6]. 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 becomes untenable. Some of the specific challenges associated with VCA are:

1) Feature expansion beyond validated functionality: As a software product becomes widely used, features are often added that are not covered by validation testing. These features 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.

III. Current Baseline State for TriBITS projects

Before describing the new TriBITS lifecycle model it is worthwhile to provide some background on the current state of TriBITS-based software engineering and development practices (e.g. as used by Trilinos) in order to 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.
- 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 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 as `-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 [5] and [6]). Therefore, the partitioning of software into PS, SS, and EX sets does not directly address software quality issues.

IV. 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 V and [7] (regulated backward compatibility is not described in this paper due to lack of space), respectively.

[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.

Defined: PS, SS, and EX tested code:
PS (Primary Stable) tested code has minimal outside dependencies 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 (Third Party Libraries) 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 an 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.

The proposed lifecycle phases / maturity levels / classifications for the TriBITS Lifecycle Model are:

1) Exploratory (EP):
   - Primary purpose is to explore alternative approaches and prototypes, not to create software.
   - Generally not developed consistent with Lean/Agile.
   - Does not provide sufficient unit (or otherwise) testing to demonstrate correctness.
   - Often has a messy design and code base.
   - Should not have even "friendly" customers.
   - No one should use such code for anything important (not even for research journals results, see below).
   - Generally should not go out in open releases (but is allowed by this lifecycle model).
   - Does not provide a direct foundation for creating production-quality code and should be put to the side or thrown away when starting product development (see [7] for a more in-depth discussion of the transition of Exploratory code to Research Stable code).
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 the idea of *aggregate course-grained unit tests* in [7]) but it may be big change for a typical programmer.
   - Has a very clean design and code base maintained through Agile practices of emergent design and constant refactoring [8].
   - Generally does not have higher-quality documentation, user input checking and feedback, space/time perfor-

mance, portability, or acceptance testing.
- Would tend to provide for some regulated backward compatibility but might not.
- Is appropriate to be used only by "expert" users.
- Is appropriate to be used only in "friendly" customer codes.
- Generally should not go out in open releases (but 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 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 the good qualities of Production Growth code.
- Primary development includes mostly bug fixes and performance tweaks.
- Maintains rigorous backward compatibility with typically no deprecated features or 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 and maintains very a clean and 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 V) and are essential for the transition to eventual production quality. Acceptance testing requires end-user applications, so acceptance testing naturally starts at a low level, then increases as more users accept the software and add (automated) acceptance tests. More formal application validation testing [9] would be categorized as acceptance testing in this lifecycle model.

The more user-focused quality aspects of production software, including 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 rather 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, resulting in non-sustainable software. Therefore, the foundational elements of strong testing and clean 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.

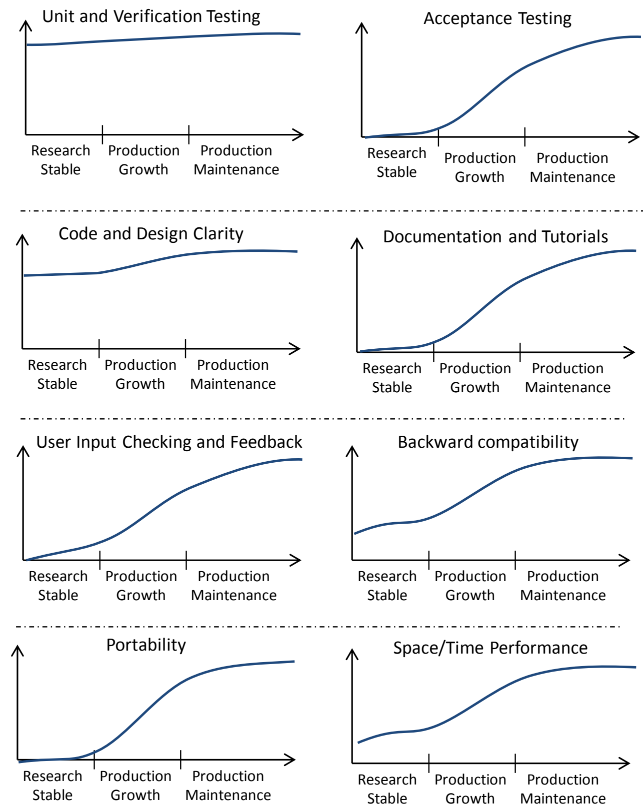

Fig. 1. Typical levels of various production quality metrics in the different phases of the proposed Lean/Agile-consistent TriBITS lifecycle model.

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 in peer-reviewed 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 in cases where the development team chooses to opt out of the TriBITS lifecycle model.

Before describing other aspects of the TriBITS lifecycle model in more detail, the important concept of self-sustaining software is defined in the next section. Regulated backward compatibility, another critical concept, is not discussed in this paper due to lack of space, but is defined and discussed in great detail in [7].

V. 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 [10]. 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 [11].
- *Exceptionally well tested*: 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 external upstream software dependencies 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, 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, including critical porting work. Alternatively, customers of non-open-source commercial software must rely on the supplying vendor for porting and needed modifications, which they might not be able to do for various reasons. What is critical to a given project is not that upstream dependent software is open-source to everyone but that it is open source to their project. This can be provided through license agreements and provide the same protection to the downstream project.

A high-level document defining the scope, high-level goals, and core domain of the software is needed to maintain the "conceptual integrity" of the software [12]. This document will be short and focused and will define the high-level "Core Domain Model" for the particular piece of CSE software [11]. A good example of a domain model for a multi-physics coupling package is given in [13].

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 [14]). Code with strong unit and verification

tests should have very few defects (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) [15]). 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 producing trusted production-quality software. This is probably the biggest weakness of the commonly used validation-centric model described in Section II.

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 or has too many entangling dependencies and tricky behaviors, the software will not be able to be affordably maintained, ported to new platforms, or changed for other purposes. Preserving a clean internal structure while developing software is most efficiently done using continuous refactoring and redesign [5]. This process must be skillfully and rigorously applied during maintenance and further development, or the result will be the slow death of "software entropy", making the code unsustainable [12]. 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 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 TPL 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.

Satisfying the basic properties of self-sustaining software is not sufficient. It is also critical that the upstream dependencies of self-sustaining software be self-sustaining. 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. (Note that depending on standard ubiquitous interfaces like the C++ standard library classes or MPI is not considered a risk and does not require access to source code or self-sustaining software.)

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.

Note that self-sustaining software does not necessarily have good user documentation, or any user-oriented examples, or necessarily produce good error messages for invalid user input. While these properties are important for the adoption and usage of any piece of software, they do not affect the sustainability of the software for existing client projects.

VI. 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. This applies to packages in all of the TriBITS phases. 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 (making the code 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 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.

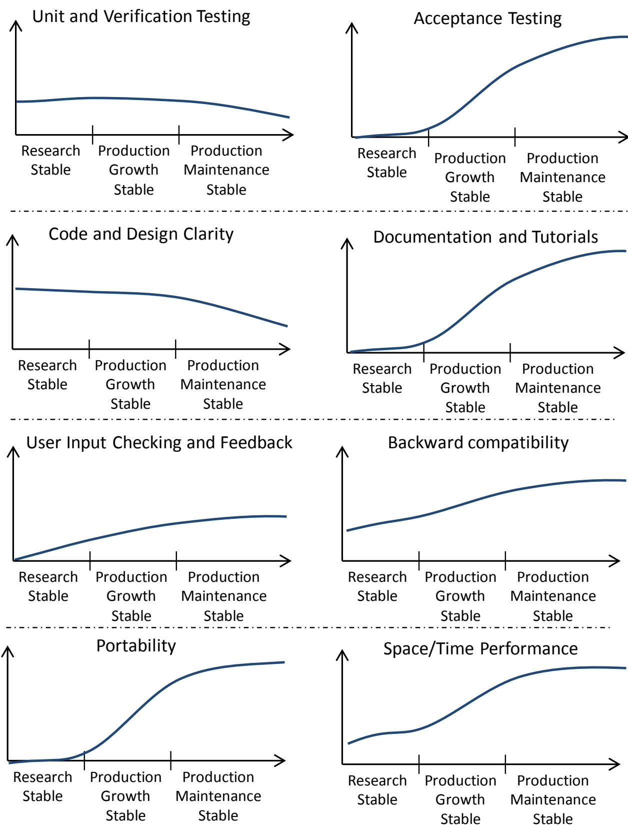

Fig. 2. Example of the more typical variability in key quality metrics in a typical CSE software development process.

VII. Comparison to a Typical CSE Lifecycle Model

A more typical non-Lean/Agile software development process used to develop CSE software (as determined by personal experience and a number of studies and also consistent with the VCA) would suggest that its quality metrics are like those shown in Figure 2, 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 [12].

Note that the timelines for user-oriented metrics, including documentation, acceptance testing, user input validation, backward compatibility, portability, and performance, 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 require clean code or good testing.

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).

VIII. 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 self-sustaining software for even Research Stable code for one or more reasons (i.e. lack of tests, messy code, troublesome external dependencies). However, some of these established packages are highly useful (see Section VI). Hardening has occurred through extensive use by customers and bug fixes, and active development has ceased. For certain common use-cases, these packages are well validated (see Section II). This is a typical way in which testing and hardening is performed in CSE and other domains but is inconsistent with Lean/Agile and does not lead to self-sustaining software.

This code is useful and represents important capabilities, so it needs to be maintained and improved. However, despite its usefulness, such software must be considered "Legacy Code" because it lacks sufficient unit and verification tests (as per definition in [14]). Therefore, the current generation of packages will be grandfathered into the new lifecycle model 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 [14] 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. All the time, be rebuilding and rerunning all the affected unit tests to make sure changes don't break the existing behavior of the code.

5) *Refactor*: Refactor the code that is now being adequately tested in order 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 repeatedly in small chunks over a long period of time, even the worst legacy software (i.e. 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.

By applying the powerful incremental refactoring and testing approaches described above, a piece of software that might otherwise be considered hopeless may actually 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 [14] and the Netscape 6.0 experience[2]).

When development teams do not agree to use the Agile Legacy Change algorithm to modify their legacy packages, those packages should be excluded from the lifecycle model and categorized as Unspecified Maturity.

IX. Summary and Next Steps

The CSE community needs to change the way research-based CSE software is developed. 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 [16]). 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 the typical journal peer-review process does not do this).

The next step for most software projects serious about trying to adopt and adapt the TriBITS lifecycle model is to define processes and standards in order to implement it within the project. Projects already using the TriBITS build and test system will have the easiest time but other projects can implement this lifecycle model without using the actual TriBITS system. The first projects that will try to adopt the TriBITS lifecycle model will be Trilinos and the DOE CASL VERA project.

Finally, as mentioned in Section VIII, existing packages will necessarily need to be “grandfathered in”. This newly defined TriBITS lifecycle process will not magically get everyone who develops software to do so 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 various CSE projects to where they need to be with respect to software quality. The potential benefit of building a large ecosystem of CSE software using this type of lifecycle model is huge and could significantly accelerate progress being made in CSE by allowing the best software from the best experts in many complex disciplines to be integrated together to solve the hardest problems in CSE.

Acknowledgment

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

References

[1] J. M. Willenbring, M. A. Heroux, and R. T. Heaphy, “The Trilinos software lifecycle model,” in *ICSEW ’07: Proceedings of the 29th International Conference on Software Engineering Workshops*. Washington, DC, USA: IEEE Computer Society, 2007, p. 186.

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

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

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

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

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

[7] R. A. Bartlett, M. M. Heroux, and J. M. Willenbring, “Tribits lifecycle model: Version 1.0 (a lean/agile software lifecycle model for research-based computational science and engineering and applied mathematical software),” Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, Technical report SAND2012-0561, 2012.

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

[9] 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,” Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, Technical report SAND2005-3662P, 2005.

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

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

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

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

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

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

[16] G. Miller, “A scientist’s nightmare: Software problem leads to five retractions,” *Science*, vol. 314, no. 5807, pp. 1856–1857, 2006.

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