A Rally Software Development Corporation Whitepaper

Agile Software Development with Verification and Validation in High Assurance and Regulated Environments

By Dean Leffingwell

Abstract: In the last decade or so, Agile software development methods have proven their worth in a variety of industry settings, delivering faster time to market, increased productivity, higher quality, and improved morale and motivation. Traditionally, however, these methods have not been applied to high assurance and regulated environments, those industries where the economic or human cost of errors is unacceptably high. There, enterprises have relied primarily on sequential, stage-gated, waterfall methods, often meeting verification and validation requirements via burdensome documentation and labor intensive, and potentially error prone, manual processes. However, many such enterprises have concluded that in order to achieve the next level of product quality and safety improvements, not to mention enhanced competitiveness, adoption of a more agile approach is required. In this whitepaper, the author describes an Agile software development approach for high assurance systems that addresses many of the challenges found in these environments.

The whitepaper concludes with an appendix that provides some examples of high assurance Agile tooling automation via Rally Software's Agile Application Lifecycle Management Platform. I owe a special thanks to Craig Langenfeld and Yvonne Kish of Rally Software for their substantial contribution to this work, as well as to the high assurance Agile development practitioners who contributed to the examples.

Contents

- Introduction ... 2
- Agile Crosses the Chasm to High Assurance Development ... 2
  - Introduction to Medical Device Exemplar ... 3
  - Regulations and Guidelines for Medical Device Development ... 3
    - Regulations ... 4
    - Guidelines ... 4
  - Software Verification, Validation, and Traceability ... 5
  - Software Requirements Specification ... 5
  - FDA Guidance is for Concurrent Engineering, NOT Waterfall Development ... 6
- Scaled Agile Delivery Model ... 8
- An Agile, High Assurance Lifecycle Model ... 9
  - System Intent, Vision and Product Requirements Document ... 10
- Iterating and Continuous Verification ... 11
  - Iterating ... 12
  - Define|Build|Test Teams ... 13
  - Agile Requirements Backlog Model ... 13
  - User Stories as Software Requirements Specification ... 14
  - User Story Verification ... 15
    - Traceability from User Story to Code ... 15
    - Traceability from Code to Unit Tests ... 15
    - Traceability to User Story Acceptance Tests ... 15
  - User Story Validation ... 16
- Validating System Increments ... 16
  - Validation Sprint Length ... 17
  - PRD to SRS Traceability ... 17
  - Testing Features ... 17
  - Testing Nonfunctional Requirements ... 19
  - Finalizing Documents ... 20
  - Finalizing Traceability Matrices ... 21
- Conclusion ... 21
- Bibliography ... 23
- About the Author ... 23
- About the Primary Contributor ... 24
- Appendix ... 25

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Introduction

Over the last two decades, the movement to iterative, and now *Agile*, development methods has been one of the most important trends affecting the software industry. In *Scaling Software Agility* [Leffingwell, 2007] and Agile Software Requirements [Leffingwell, 2011], I describe how Agile methods left the realm of small software teams and programs and are now being applied at enterprise scale. As the methods were relatively new (er) at the time, I highlighted many of the documented benefits, including increases in productivity, quality, and team morale and job satisfaction. In turn, Agile software enterprises receive the ultimate beneficial result – improved economic outcomes, higher safety, value, convenience, efficiency, efficacy, and indeed, better standards of living for society at large – that accrue when we improve on this strange mix of art, science, and engineering that we call software development.

When it comes to software quality, it comes as no surprise to Agile proponents that, when properly implemented, Agile development achieves much higher quality than its waterfall predecessor. The tyranny of the urgent-defect-triage-end-game is largely mitigated. And while we will always need to address defects, non-conformances and minor user need misfires, with Agile we can focus far more on two primary things: 1) better understanding user needs, and 2) building software that has quality, including safety and security, as its *organic* basis. Given this focus, we now find Agile in another mainstream, those enterprises developing software for medical equipment, pharmaceutics, avionics, military systems, financial trading systems, and other high assurance applications. Here, the presence of any material defect may have an unacceptable human or economic cost.

In this whitepaper, we describe an Agile approach to developing high assurance software systems. As a baseline, we will explore regulations and guidance associated with the development of medical systems as regulated by the US FDA and similar international agencies. While this will be our example, many other high assurance industries have similar regulations and requirements, and most assume the need to *verify and validate* our requirements and the resultant software application to be sure that our solution works as, and only as, intended.

We describe an Agile lifecycle model that supports the business need for constant feedback and risk identification and mitigation, and lessens the dependencies on abstract, document-oriented milestones. In its place, we will use working code as the basis for feedback and decision-making. We suggest verification and validation activities and artifacts that support our need to make sure the system works only and exactly as we intended, and simultaneously provides traceability mechanisms we need to demonstrate these facts to other stakeholders, including regulatory bodies. The appendix provides examples, screenshots and explanations for various reports and artifacts that we can generate using Agile tooling solutions such as Rally Software's Agile Lifecycle Management Platform, HP's Quality Center, and Subversion version control systems.

Agile Crosses the Chasm to High Assurance Development

Before we proceed, it is important to know that there is already substantial evidence of successful adoption Agile development methods in high assurance industries. For example, in *Adopting Agile in an FDA Regulated Environment* [Abbott 2009], Abbot Labs describes how they applied Agile development in the development of a nucleic acid purification platform in its Molecular Diagnostics business. They also took the time to compare the Agile method to a similar predecessor project built with prior, waterfall practices. The conclusions are compelling:

> *(On the new project)*: Fewer defects were found…the availability of working software, early on was a significant factor.
>
> *(By comparison to the prior project)*: We estimate that using an agile approach … would have resulted in the overall project duration being reduced by 20-30%...a headcount reduction of 20-30%... and a net cost savings of 35-50%.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

> *(In conclusion)*: *This experience has convinced us that an agile approach is the approach best suited to development of FDA-regulated devices.*

As further evidence of this trend, the Association for the Advancement of Medical Instrumentation (AAMI) has recently released a Technical Information Report, which provides extensive recommendations for complying with international standards and FDA guidance documents when using Agile practices to develop medical device software[1].

Agile development is also making significant inroads into other high assurance industries, including none other than the US Department of Defense. David Rico commented on a recent AFEI[2] DoD Agile Development Conference dedicated to promoting Agile acquisition and IT development practices:[3]

> It reinforced the U.S. DoD's commitment to the use of Agile Methods. Furthermore, it was interesting to see that Agile Methods are in widespread use by the U.S. DoD, and that no individual organization, project, group, or person is practicing them in isolation. Prior … both the commercial industry and DoD contractors believed the U.S. DoD was not committed to Agile Methods, which is an enormously incorrect assumption…..It's a popular urban legend that the DoD uses (only) traditional methods such as DoD 5000, CMMI, PMBoK, and other waterfall-based paradigms to develop IT-intensive systems. The AFEI DoD Agile Development Conference shattered that myth completely.

Introduction to Medical Device Exemplar

The domain of high assurance development is extremely broad, and regulations and interpretations vary from industry to industry. In this whitepaper, we use medical device development under the auspices of the US Food and Drug Administration as an example regulatory environment. In our experience, the expectations for practices and compliance in this industry are quite similar to many others, so if we understand this one instance, then we might be able to understand something about them all. However, the following disclaimer is appropriate:

> *Regulatory requirements for software differ from industry to industry, and interpretations and enforcement practices are constantly changing. Any suggestions provided in this whitepaper do not constitute legal or regulatory advice. Information in this whitepaper should only be applied by qualified professionals with direct knowledge of their specific industry.*

With that disclaimer out of the way, let's look at regulations and guidance for development of medical devices under the auspices of the US FDA.

Regulations and Guidelines for Medical Device Development

Each industry has its own thicket of regulatory bodies and published standards that must first be parsed for understanding. Figure 1 describes a chain of both *regulatory* and *guidance* documents that govern medical device software development as provided for under the US FDA Code of Federal Regulations (CFR) 21.[4]

---

[1] The AAMI Technical Information Report TIR-45: Guidance on the Use of Agile Practices in the Development of Medical Device Software: Info can be found at http://my.aami.org/store/detail.aspx?id=TIR45-PDF.

[2] The Association for Enterprise Information (AFEI). See http://www.afei.org/Pages/default.aspx.

[3] See http://davidfrico.com/afei-2010.doc

[4] Note: while this post and thread focuses on *US FDA* Good Manufacturing Practices (GMP) requirements for medical devices, these requirements are generally harmonized with International Organization for Standards (ISO) 9001:1994 and ISO DIS 13485.

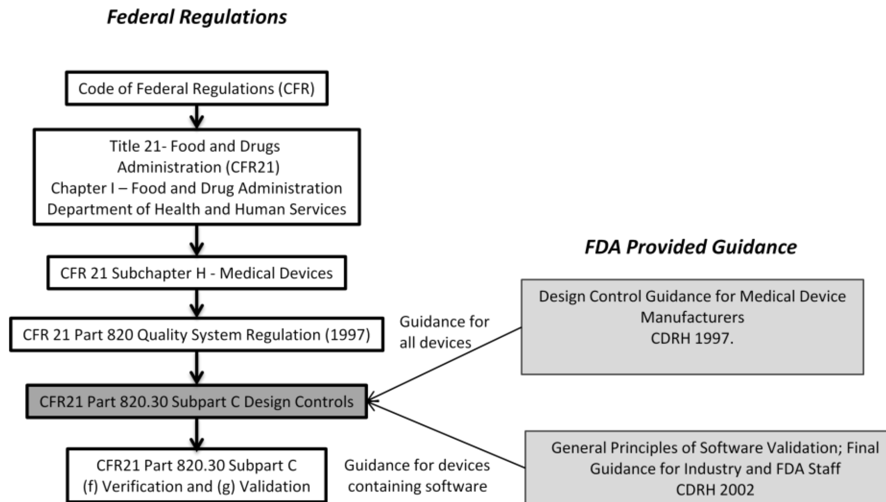


Figure 1 Regulatory and guidance documents for US medical device development

A summary explanation of these documents is provided in the paragraphs below.

Regulations

In our case, the law starts with the US Code of Federal Regulations (CFR), which is the law of the land in the U.S. Title 21 of the CFR, is reserved for rules of the Food and Drug Administration (FDA). Here, we find the following additional "sub" regulations.

CFR 21 Subchapter H – Medical Devices. Covers the general design, manufacture and marketing of medical devices.

CFR 21 Part 820 Quality System Regulation. Defines specific regulatory requirements for the design and development of such devices, intended to ensure that finished devices will be safe and effective and otherwise in compliance with the Federal Food, Drug, and Cosmetic Act.

Quality System Regulation, CFR21 Part 820.30 Subpart C Design Controls. Subpart C specifies the regulations for device design (including software development in our case). Regulatory requirements for device *verification* and *validation* are included here.

Guidelines

CFR 820.30 is the regulation that covers all medical devices, whether they include software or not. It is remarkably short (just 2 pages!) and provides lots of leeway to device manufacturers. It is important to note that this document, at the bottom of the chain, *contains all of, and the only specific federal regulations* which govern medical device development.

However, as an aid to FDA staff and the industry, CDRH has published additional *guidelines* that add specificity to how these regulations will be interpreted. First, for 820.30, Design Controls, there is the publication *Design Control Guidance for Medical Device Manufacturers*, [FDA CDRH 1997]. It is important to note that these guidelines are just that, guidelines, and they do not have the force of regulation. However, they do set expectations on the part of the industry and regulators as to how the regulations should be generally interpreted.

And finally, and more specifically to our context, CDRH has published a document which provides guidelines for the general principles of software validation, *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*, [FDA CDRH 2002].

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Software Verification, Validation, and Traceability

CFR 21 Part 830 Subpart C Design Controls, Paragraphs (f), (g) mandate device design verification and validation. In General Principles of Software Validation [FDA CDRH 2002], we find:

> Software *verification* provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase. Software verification looks for consistency, completeness, and correctness of the software and its supporting documentation, as it is being developed, and provides support for a subsequent conclusion that software is validated. In other words, verification ensures that "you built it right."

> Software *validation* is confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled. Since software is usually part of a larger hardware system, the validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements. In other words, validation ensures that "you built the right thing."

In addition, the guidelines go on to describe *traceability* and *traceability analysis* as primary mechanisms to assure that verification and validation are complete and consistent, and that a traceability matrix is the documentation record that provides the objective evidence of the completeness of the verification and validation. For definitions here, we refer to FDA Glossary of Computer Systems Software Development Terminology[5]:

> Traceability. (from IEEE) (1) The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; e.g., the degree to which the requirements and design of a given software component match. (2) The degree to which each element in a software development product establishes its reason for existing.

> Traceability Analysis. (from IEEE) The tracing of (1) Software Requirements Specifications requirements to system requirements in concept documentation, (2) software design descriptions to software requirements specifications and software requirements specifications to software design descriptions, (3) source code to corresponding design specifications and design specifications to source code. Analyze identified relationships for correctness, consistency, completeness, and accuracy.

> Traceability Matrix**.** (from IEEE) A matrix that records the relationship between two or more products; e.g., a matrix that records the relationship between the requirements and the design of a given software component.

Software Requirements Specification

In addition to these definitions, this document [FDA CDRH 2002] also provides an introduction and definition to another important artifact:

> A documented *software requirements specification* (SRS) provides a baseline for both validation and verification. The software validation process cannot be completed without an established software requirements specification (Ref: 21 CFR 820.3(z) and (aa) and 820.30(f) and (g)).

From the guidance, such a document typically contains:

- All software system inputs, outputs, and functions
- All performance requirements, including throughput, reliability, accuracy, response times (i.e., all nonfunctional requirements)

[5] see FDA Glossary of Computer Systems Software Development Terminology on line at http://www.fda.gov/iceci/inspections/inspectionguides/ucm074875.htm

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

- Definition of external and user interfaces
- User interactions
- Operating environments (platforms, operating systems, other applications)
- All ranges limits, defaults and specific values the software will accept
- All safety related requirements specifications, features, or functions implemented in software

Clearly, the verification and validation (V&V) aspects of the regulations essentially mandate the creation a software requirements specification[6]. In contrast, non-high assurance Agile development teams don't typically create these in a formal way; instead they use the backlog, collections of user stories and acceptance criteria, persistent test cases and the code itself (i.e.: with defined templates for auto insertion of code header information and checkin code comments) to document system behavior. However, in this context, it is clear that we will need to develop and maintain a software requirements specification, as we simply cannot do V&V without it. However, the fact that we need such a document *doesn't mandate that we do it all Big Bang and Up-Front.* To be agile, we can and will develop it incrementally with the artifacts incrementally generated within agile teams. The benefit here on agile teams is that the documentation is continuously updated with the agile team artifacts, so documentation is not done after the fact.

FDA Guidance is for Concurrent Engineering, NOT Waterfall Development

[FDA CDRH 2002] (40 pages) provides the most specific guidance applicable to the validation of medical device software. *It is important to note, that neither this document, nor CFR820.30 itself, constrains development to single pass, stage-gated, waterfall activities*.

> From General Principles of Software Validation….. [FDA CDRH 2002]: This guidance recommends an integration of software life cycle management and risk management activities. *While this guidance does not recommend any specific life cycle model or any specific technique or method*, it does recommend that software validation and verification activities be conducted throughout the entire software life cycle.
>
> From Design Control Guidance for Medical Device Manufacturers [FDA CDRH 1997]: Although the waterfall model is a useful tool for introducing design controls, *its usefulness in practice is limited*… for more complex devices, a concurrent engineering model is more representative of the design processes in use in the industry.
>
> Even more specifically, IEC62304, (a widely recognized international standard for medical device software which is largely harmonized with FDA's interpretation of CFR 820.30) states: … these activities and tasks may overlap or interact and may be performed iteratively or recursively. *It is not the intent to imply that a waterfall model should be used.*

Notwithstanding the above, and whether we are doing high-assurance development or not, it is likely that our enterprise's standard approach to software development followed the over-simplified waterfall model. After all, that's what most of us have done for years, so why would the high assurance development teams be any different? To be flippant for a moment, perhaps we headed down this path because it's simply easier to draw. Actually, we have done this for years because the linear waterfall SDLC phase gates are so easy to map to milestones for management and they fit in so neatly with linear management forecasts and budgets for formal phase gate reviews (i.e., SRR, PDR, CDR, TRR, etc.) as seen in Figure 3 below. For example, in describing the application of design controls, [FDA CDRH 1997] provides the following diagram.

[6] For further description of the contents of an SRS, refer to IEEE Recommended Practice for Software Requirements Specifications and Managing Software Requirements: A Unified Approach [Leffingwell and Widrig 1997].

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

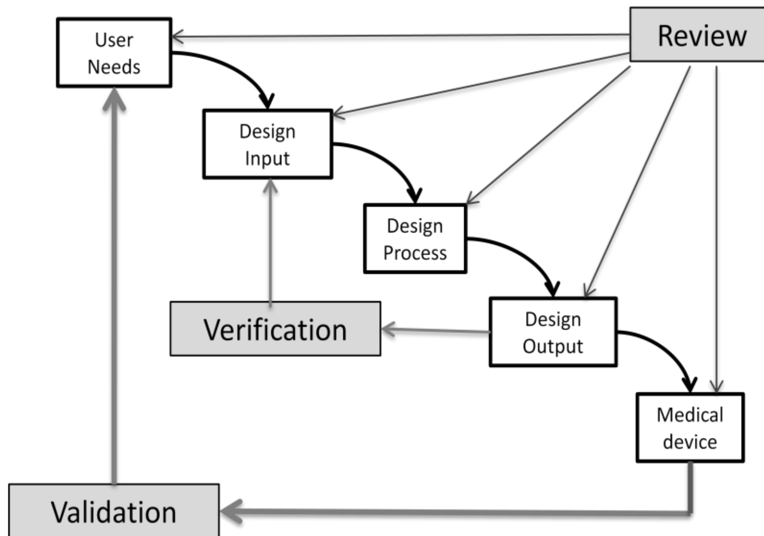


Figure 2 Application of design controls to waterfall design process. From [FDA CDRH 1997] Design Control Guidance for Medical Device Manufacturers

Most readers will recognize Figure 2 as being typical of more generic, traditional waterfall software development process models, such as that pictured in Figure 3[7].

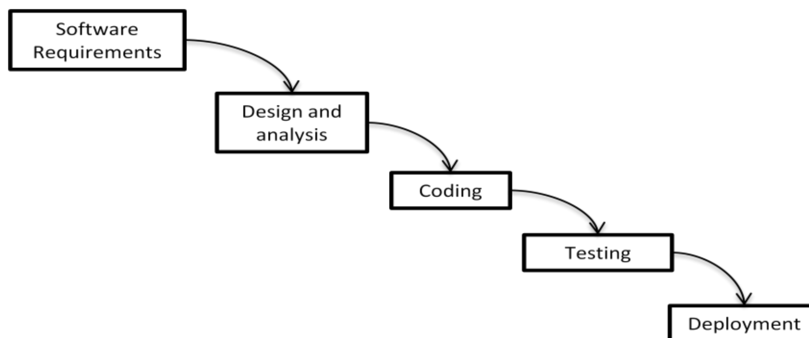


Figure 3 A traditional waterfall model view of software development

In addition, V&V activities are often aligned with a traditional waterfall approach depicted above with the commonly known V-model depicted in Figure 4 below. In the V-model, each phase of development is aligned with the type of V&V activities, which heavily includes testing, for each phase.

[7] Note to the software historians: Winston Royce's seminal 1970 article, which initially described the sequential model which came to be known as the waterfall model, included the following statement "I believe in this concept (analysis before coding etc) , but the implementation above (a sequential, single pass, reqs-analysis-design-coding-testing) is risky and invites failure." In other words, he concluded that the waterfall model, as we often imply today is NOT a recommended model for development.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

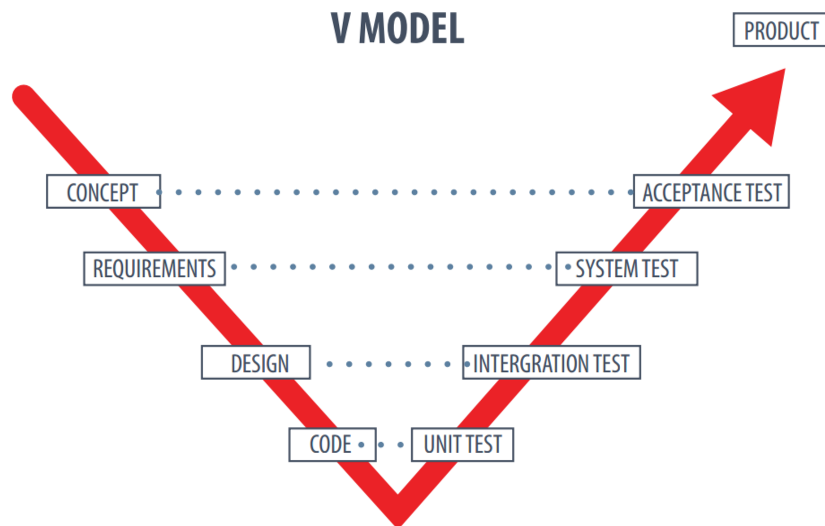


Figure 4 A traditional V Model view of software development phases aligned with V&V and testing phases

Such diagrams, familiar to us all, offer some positive attributes, at least on the surface:

- It is seemingly simple and logical (what could be more logical than code following requirements, and tests following that?)
- In theory, you only have to do "it all" once (especially verification and validation, which can be both labor intensive, expensive and error prone).

Of course, we know it doesn't work well that way, and that's why we strive for agility in the high assurance markets, just like we do everywhere else. However, for many, the apparently beguiling (but false) simplicity of the model is one reason that it has made its way into the various governing documents, milestone reviews, corporate quality standards, etc.

Let us be clear: with respect to this one industry, and with respect to these specific guidelines, *any notion that we are mandated to apply a single-pass, waterfall model to software development is an industry myth, one which has likely been perpetuated by our own waterfall past ("we have always done it this way") and our existing quality management systems, and not because "the regulations make us do it".*

Scaled Agile Framework

As we noted earlier, many enterprises have already moved to a scalable Agile development model. Base don these experiences in the field, we have evolved a full-fledged, public-facing framework, which can be found at ScaledAgileFramework.com. The Frameworks "Big Picture" serves as its UI and navigation paradigm, as Figure 5 illustrates:

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

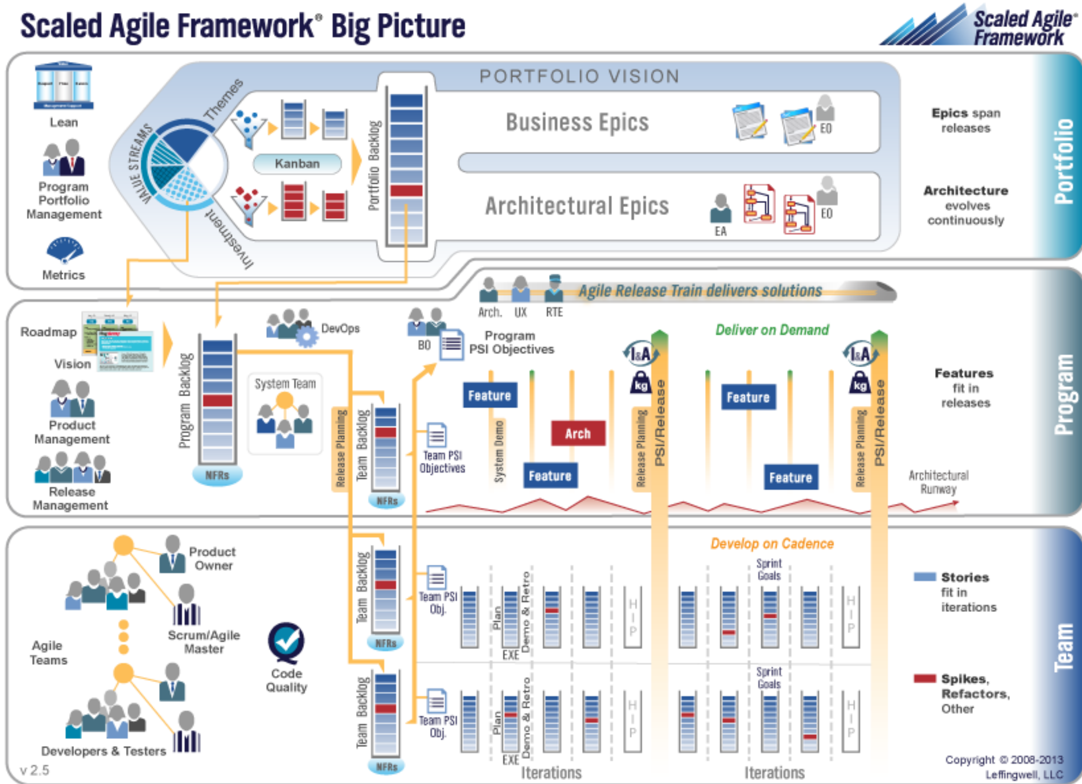


Figure 5 Big Picture of the Scaled Agile Framework.
Source: ScaledAgileFrameowrk.com

This framework has a number of interesting attributes, many of which we will be able to apply directly to our high assurance context:

- Software development proceeds in a short series of *iterations* (or Sprints)
- For requirements, Agile teams work against their local, project *backlog*, which consists primarily of *user stories* that are written just prior to the iteration in which they are implemented
- Teams break stories down into tasks, which is a fine grained work breakdown structure (which we will put to good use shortly)
- Teams work against a common program backlog, comprised of a set of features which describe the system intent, or solution *Vision*
- Periodically, the teams collaborate on a HIP sprint, used to eliminate any remaining technical debt, perform full system integration and regression testing, and to finalize documentation
- At the end of each HIP sprint, the program has produced a release (or Potentially Shippable Increment) of the full software asset stack.

An Agile, High Assurance Lifecycle Model

The Agile model above is robust and scalable, and has been successfully applied in a number of industries. However it does not assume the development of specific requirements documents, nor explicit verification and validation, so we extend this a bit in the high assurance context. With a nod to the model above and the Abbot Labs whitepaper [Abbot 2009], and with due respect to the necessary verification and validation

activities, we offer Figure 6 as a model for Agile, iterative and incremental development in high assurance markets:

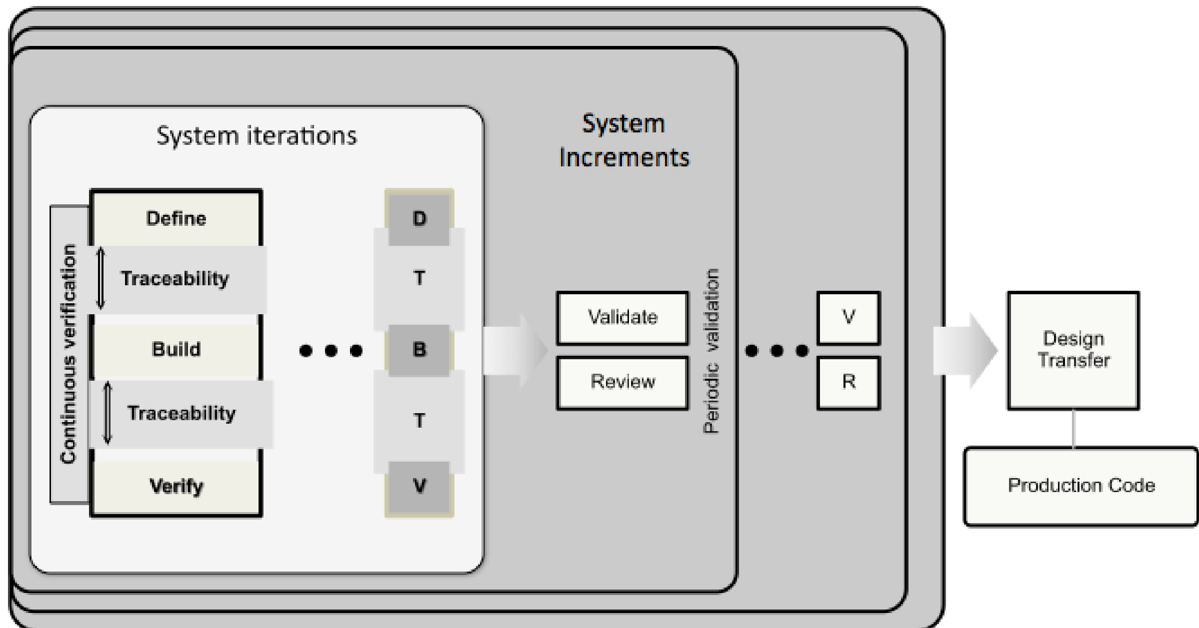


Figure 6 An Agile, high assurance lifecycle model

In this model, we use a series of short *system iterations*, each of which defines, builds and *verifies* some new and valuable user functionality. Within each iteration, we will define some new requirements and write and test the code that fulfills them. We will also perform continuous verification, including inherent traceability, and automate these activities wherever possible.

Periodically (typically after 3-4 iterations) we will run a validation (HIP) iteration to v*alidate* a new system *increment*, so that we can be assured we still meet the full system requirements, and not just the ones we have implemented in the most recent iteration. In addition, we will have to conduct a *review* prior to our making the system available for testing in the actual usage environment, and eventually, for a full product release.

System Intent, Vision and Product Requirements Document

Previously, we have described the SRS, which is prescribed by regulatory guidance and is the center for most software development activities. However, this document does not stand-alone. For most systems, including systems of systems and devices that contain both hardware and software, the governing document, which resides above the SRS is usually a Product Requirements Document (we will use the acronym PRD, but other names, Marketing Requirements Document, System Specification, etc. are used as well). The PRD contains the complete specifications for the device as a whole.

From the perspective of the market, management and the FDA, the PRD is probably the most critical document because it describes exactly what the system is intended to do for the users. Keeping the level of abstraction sufficiently high enough to make sure the document is understandable, and yet specific enough to drive development is the art of good product definition. The PRD typically contains at least these major elements:

- statement of device purpose
- system context diagram

- descriptions of users and operators
- features of the device
- nonfunctional requirements
- operation and maintenance requirements
- safety features and hazard mitigation

While some of these items are specific to medical devices, guidance for the content of the PRD closely mimics the generic "Vision" document from the Rational Unified Process and other, similar methods. See footnote[8] for additional information.

Since this document describes the intent of the system, it's just as important as the SRS, and developing and maintaining this document is a significant program activity. In a fashion similar to the SRS, it may be developed somewhat incrementally. However, unlike the SRS, which is really a response to system intentions, much of the content of the PRD will be developed in advance of the bulk of development, as it states what the device is intended to do.

Fortunately, making the PRD understandable by managing the level of detail, and simultaneously not over constraining Agile software development with over-specificity are sympatric goals, so keeping the descriptions high level in the PRD serves both purposes.

In any case, from the standpoint of system behavior, the primary content of the PRD is the set of *features* (high-level descriptions of system behavior that addresses a user need) the system provides to deliver its efficacy and safety. Features can be expressed in user voice form (see later) but are more typically expressed in a short keyword phrase or sentence. For example, an EPAT[9] device might include:

```
Pulse wave frequency is adjustable from 1-21 hertz

Pressure amplitudes can be varied from 1-5 bar

Acoustic energy may be concentrated to a focal area of 2- 8 mm in
diameter
```

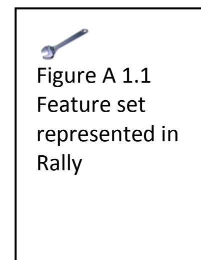


Figure A 1.1 Feature set represented in Rally

While the PRD makes the "labeling claims" for the device via these features, the actual work of implementation is left to the software (and hardware, of course) so part of our verification effort is to assure that every feature traces to one or more user stories (or other forms of software requirements expression).

The traceability matrix is an important document that will provide the evidence needed for forward and backward traceability for verification and validation from requirements to test, and completeness of test back to requirements. It is important to include the product requirements in the traceability matrix for completeness to show product to software requirements traceability, then to trace through to design, code and test.

Iterating and Continuous Verification

With this background behind us, we can now move forward with suggestions for how to implement the high assurance agile lifecycle model illustrated in Figure 5. Thankfully, the bulk of the work is focused on developing and verifying the actual software that is created to achieve the desired performance of the system under construction. Most of that is coding and testing activity, which we will approach using rigorous, but fairly standard Agile development practices including continuous integration, peer review/pair programming,

[8] For a more traditional representation, see the Appendix in *Managing Software Requirements: First Edition, A Unified Appro*ach [Leffingwell 1999] towards a more Use Case-driven view in *Managing Software Requirements: Second Edition, A Use-Case Approach* [Leffingwell 2003] and a more agile version in *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise* [Leffingwell 2011].

[9] Extracorporeal Pulse Activation Therapy Device. (A device that appears to have done wonders for my chronic Achilles tendonitis.)

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

unit testing, Test-Driven Development, automated build and verification systems, coding standards, regular reviews and retrospectives. We will also keep in mind the need for continuous verification, and we will automate as much of that as possible using appropriate tooling.

Iterating

The basic pattern in agile development is the iteration (or Sprint in Scrum). Each iteration has a fixed length (often two weeks) and the pattern for the period is described in Figure 7 below.

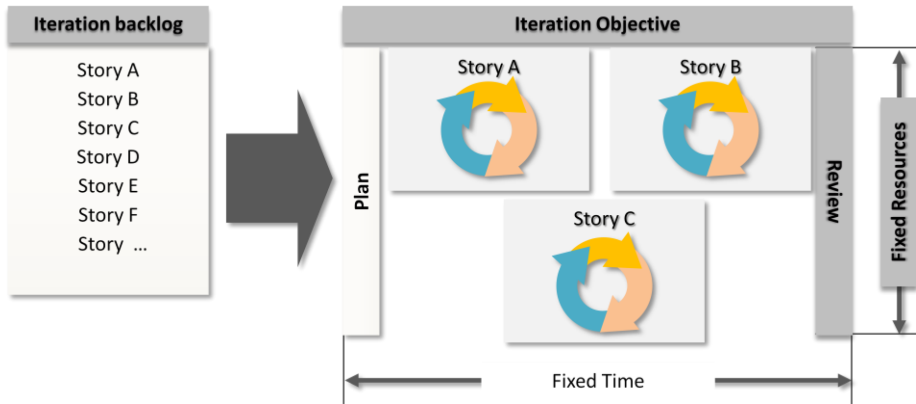
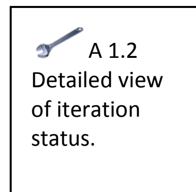


Figure 7 Basic iteration pattern

During the course of the iteration, teams first plan the iteration, breaking the stories into *tasks*, and then commit to completing some number of stories. The team then implements the stories, driving each towards a *definition of done*. The definition of done establishes policy for what constitutes story completion, which assures that the story is properly coded, reviewed, tested and accepted into the product baseline.



It is very important in a high assurance environment that the definition of done include all quality, safety, security and regulatory requirements that need to be satisfied for completeness. Furthermore, the verification and validation for definition of done is traced through to the acceptance criteria for each user story and then originating the product requirement. This will ensure that the user story is complete from a customer point of view for functionality, that all non-functional requirements are satisfied from a system point of view, and that all quality, safety, and security requirements are complete from a regulatory point of view. A well-defined definition of done attributes to a proactive preventative approach to quality and risk management.

During the course of the iteration, teams meet daily to discuss progress and address issues. Any potential risks are identified, tracked and raised as impediments, and risks that have been resolved are reported and closed. Progress is tracked using burn down charts and iteration status reports.

At the end of the iteration, the team conducts a demo and retrospective. The demo is used to show all stakeholders the current state of the solution, highlighting the new behaviors implemented in the current iteration. Thereafter, the teams hold a brief retrospective focusing on what improvements they can make to the development process in the next iteration. For high assurance teams, risk management and how well risks were identified and mitigated are always part of the retrospective discussion.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Define|Build|Test Teams

Doing all this in a short timebox is no trivial feat, and Agile teams are designed specifically to make this feasible. In *Scaling Software Agility* [Leffingwell 2007], we described a common Agile team model, based primarily on Scrum. We called that the Define|Build|Test to make it clear that they must have all these skills resident with the team. Here, our Define|Build|Test teams consists of:

- A product owner who has the responsibility to define and manage the backlog (upcoming requirements). In our example, this person must also have the requisite medical domain knowledge and authority to make critical decisions on the safety and efficacy of the product.
- Developers who implement both the code and unit tests that test the code
- Testers who develop and execute acceptance tests to assure that the system meets its requirements and that all acceptance criteria has been fully met
- (and most likely in high assurance environments) quality assurance team members who assist with verification, validation, traceability and other necessary documentation activities to ensure regulatory standards are met.
- A ScrumMaster who is a facilitator and coaches the team to ensure they are implementing the scrum process correctly. Also ensures that any impediments reported by the team are quickly resolved to unblock the team as soon as possible.

We mention the cross-functional team construct here because it is assumed in this model (and in all Agile development). However, for a high assurance company transitioning from waterfall to Agile development, we recognize that this step alone may be a big challenge, as it breaks down the functional silos and reorganizes along value delivery lines. Without this step, real agility cannot likely occur. Also, because this is *how we do what we do*, formalization of the team construct may be important enough to be called out in the company's quality management system (QMS) practice guidelines.

Agile Requirements Backlog Model

While Agile development is often mis-perceived to be informal and loosely structured, this is just another myth that inhibits use in serious and scaled development environments. The fact is that Agile is an extremely disciplined process, and while the terms and artifacts the teams use are different, they are typically based on a tight semantic model between requirements artifacts such as user stories, and their unit tests and acceptance tests to assure the stories have been implemented properly. In *Agile Software Requirements* [Leffingwell 2011], I described a scalable Agile requirements model, which we can now leverage directly in our high assurance context. Relevant portions of that model appear in Figure 8 below:

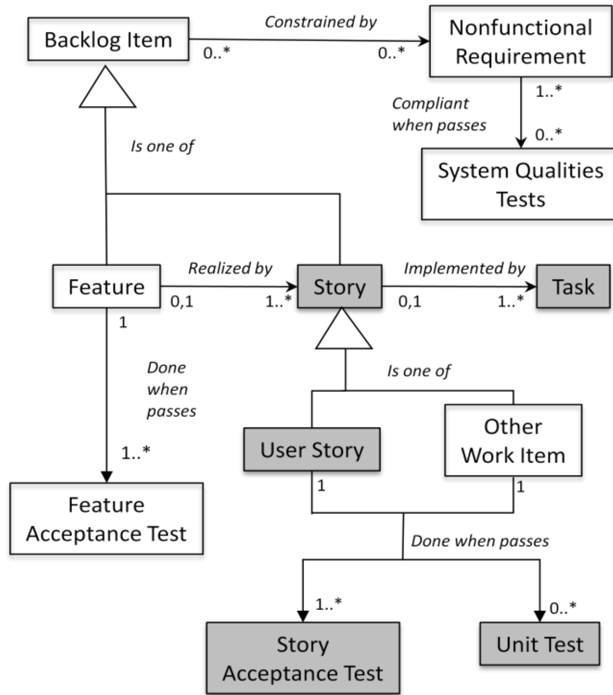
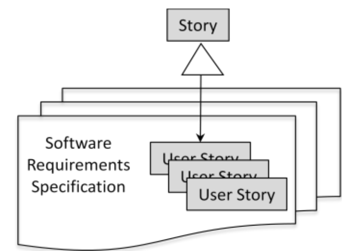


Figure 8 Scalable Agile requirements backlog model

User Stories as Software Requirements Specification

For now, we are primarily concerned with the gray boxes (stories, tasks, story acceptance tests and unit tests) in this figure. As *user stories* are the primary form of requirements expression used in Agile, the SRS we will need will be an as-built compilation of user stories. In this way, the SRS will evolve as the system evolves.

Each Agile user story is a brief statement of intent that describes something the system needs to do for the user. Typically, each user story is expressed in "user voice" form as follows:

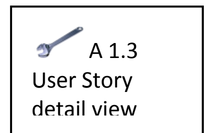


As a <role> I can <activity> so that < value>

where:

- Role – represents who is performing the action.
- Activity – represents the action to be performed
- Value – represents the value to the user (often the patient in our exemplar) that results from the activity.

For example,




```
As an EPAT (Extracorporeal Pulse Activation Technology)
technician, (<role>) I can adjust the energy delivered (<what I do
with the system>) in increments so as to deliver higher or lower energy pulses to
the patient's treatment area (<value the patient receives from my action>).
```

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

User Story Verification

User stories are sized such that they can be coded and tested within the course of a single iteration. That way, when we are done with an iteration, we have assured delivery of some new value. To assure that each new user story works as intended, we will verify it using three built-in (and semi-automated) traceability paths:

- User Story to code – path to the SCM record that illustrates when and where the code was changed to achieve the story
- Code to unit test – the path to the unit test is the "white box" test which assures the new code works to its internal specifications; linkage is maintained via the path to the SCM change set
- User Story to Story Acceptance Tests – the path to black box functional test of the user story

To do this explicitly, we can use a standard Agile tasking model as part of our definition of done for each new user story, as Figure 9 illustrates. Each task ID (maintained in our Agile project management tooling, and directly associated with the story) can be used to establish a traceability link to the specific artifact.

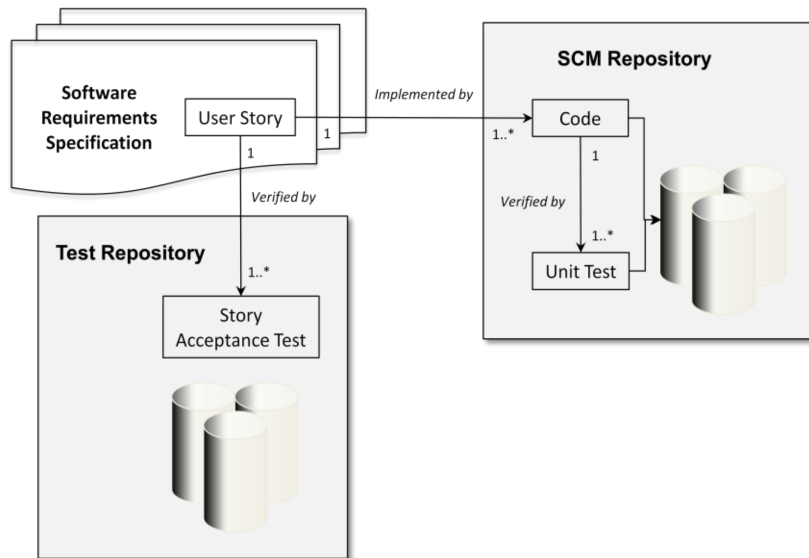
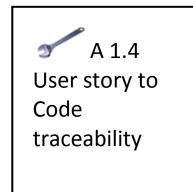


Figure 9 Explicit support for traceability to code, unit tests and acceptance tests

Traceability from User Story to Code

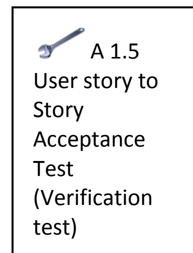


This traceability path connects the user story to the code that implements it, via the task that was used to create the code.

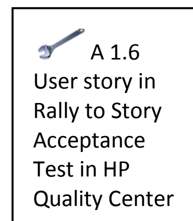
Traceability from Code to Unit Tests

The code itself is tested by the unit tests, which test the methods that were implemented to fulfill the story.

Traceability to User Story Acceptance Tests



User Story Acceptance Tests (SAT) are functional tests intended to assure that the implementation of each new user story delivers the intended behavior. SATs are usually written in the language of the business domain and are developed in a conversation between the developers, testers, and product owner. They are *black box* tests in that they verify only that the outputs of the system meet the conditions of satisfaction for user story acceptance criteria, without concern for how the result is achieved. SATs are implemented during the course of the iteration in which the story



© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

itself is implemented, as prescribed by the Definition of Done. Traceability occurs via the task connection from the user story to the story acceptance test. SATs persist in the teams testing repository tooling of choice. The successful completion of all verification and validation for all user story acceptance criteria is stored in the repository tooling as objective evidence to meet regulatory requirements.

User Story Validation

In addition, at the end of the iteration, each user story is demonstrated to the product owner and other stakeholders. This gives these authorities the opportunity to see the new user stories, comment, and if necessary, define changes for future increments. This provides a first, informal validation that the story fulfills the requirements as intended.

Validating System Increments

Periodically, we will have to validate our work prior to applying our device in any actual usage setting. In Figure 6, we illustrated multiple (typically 3 to 4) development iterations (indicated by full backlogs) followed by a HIP iteration. This pattern is arbitrary but fairly typical as it produces a Potentially Shippable Increment (PSI) (valuable and evaluate-able in our medical device setting) every 10 weeks or so.

The HIP iteration has an empty backlog, implying no new user stories. Ideally, we could ship our product every day, so the dedication of time to a hardening period is not ideal from an agile purist standpoint. But this iteration acknowledges a practical reality: development of the code itself is only part of the job, especially for systems of complexity, scale, and high cost of failure, so some work that prepares the code for release can be done efficiently outside a normal development iteration.

Therefore, the HIP sprint is a high value, dedicated time for focusing on some of these remaining activities. This can include the elimination of accumulated technical debt as well as full regression testing, traceability updates, and finalizing system documentation. Full validation for a high assurance system is no trivial effort, and will likely include at least the activities identified in Table 1 below:

System Testing	Traceability	Documentation
Resolve/close/note outstanding defects	Update and finalize all traceability matrices	Document the results of the validation
Final integration with other components and systems	PRD to SRS	Update quality systems documentation
Final regression test of Unit Tests and Story Acceptance Tests (normally accomplished in the course of each dev iteration)	PRD to Feature Acceptance Tests	Update, version control and “sign” PRD
Regression test all Feature Acceptance tests (same comment as above)	SRS to code	Update, version control and “sign” SRS
Regression test all System Qualities tests	SRS to Story Acceptance Test	Finalize user documentation
Perform any exploratory testing	Code to unit tests	Release notes and any other user guidance
Perform any/all user acceptance testing		Installation and maintenance requirements
Perform all internal acceptance (development, clinical, product marketing) testing		

Table 1 Typical high assurance HIP iteration activities

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Validation Sprint Length

Depending on the levels of test automation and other tooling, accomplishing validation in a single, two-week iteration may not be practical. In that case, some teams treat this iteration as fixed scope, not fixed length, so they take the time necessary to do a proper job. However, there is a negative impact to an open-ended time frame, as we lose the cadence and synchronization that are our primary tools to manage product development variability [see Reinertsen 2009] and the schedule may become unpredictable.

Instead, it is generally better to have an additional iteration dedicated to this purpose, or perhaps a longer one than normal. Then, if the teams fail to get the job done, they can always add another iteration for this purpose, and the "failure" will indicate where they need to invest in additional automation and documentation tooling, or address whatever other impediments remain. After all, completing critical work in a quality fashion with attention to identifying and mitigating risks in a short timebox, and working towards tight, continuous integration is what good Agile teams do.

PRD to SRS Traceability

Since the PRD is the controlling document for what the software implementation needs to do, we will need to maintain linkage and a traceability reference for the claims for the device (features in the PRD) to how those claims are implemented in software (User Stories in the SRS) in the form of a traceability matrix. Figure 10 illustrates such a set of relationships for our hypothetical device.

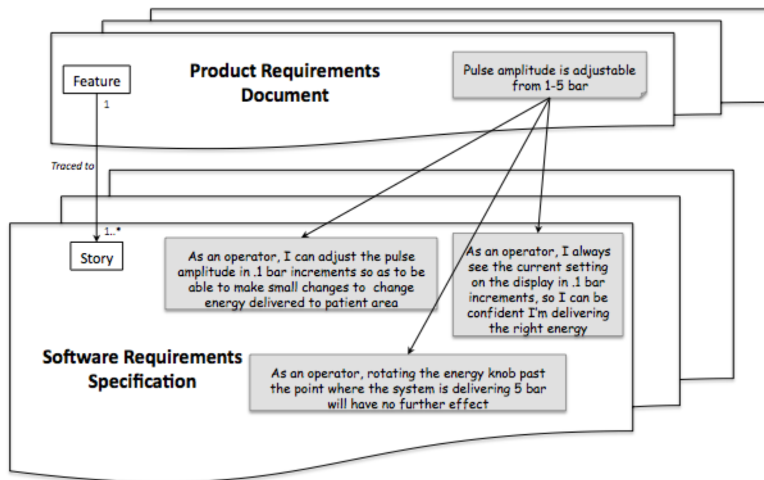
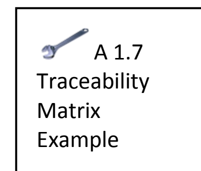


Figure 10 Example PRD to SRS Relationships for a hypothetical EPAT device

In this way we can be assured that every feature of the system traces to, or is implemented by, one or more user stories. If our analysis discovers stories that do not trace to a feature, that may be OK since all analysis does not always map to a user story, but we will need to at least understand their purpose to make sure that the system behaves only as intended. It should be documented why the analysis does not trace. This particular traceability path also gives us a way to consider the impact of potential feature level changes, and trace approved features changes into implementation by their children stories.



Testing Features

Earlier, we described the Product Requirements Document as the higher level (super-ordinate to the SRS) governing document for our device or system. Its content includes the purpose of the device, as well as a set

of features and nonfunctional requirements that describe the behavior of the system at a fairly high level. Features are primary elements of our agile requirements model, as Figure 11 illustrates:

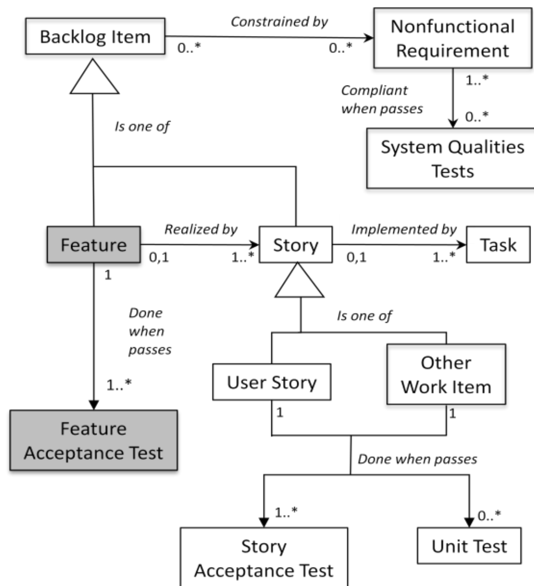


Figure 11 Features and Feature Acceptance Tests in the requirements metamodel

Since features are implemented by stories and stories are unit tested, acceptance tested and validated by the product owner, the question arises as to whether or not features must be independently tested as well. As implied by the model, the answer is yes, because:

- It can take a large number of stories to implement a single feature. Even if they all deliver value, the aggregate behavior must still be assessed.
- There are a large number of paths through the stories, which cannot be naturally tested by the independent story acceptance tests.
- Features often require integration with other systems components and features, and in many cases a single team may not even be able to access the resources necessary to fully test the feature they contributed to.

Therefore, each time a new feature from the PRD is implemented a Feature Acceptance Test (FAT) must typically be developed and persisted in the system level regression test repository. These feature tests are in addition to the story implementations and story tests, Whenever these tests are automated, typically with some custom, business logic-level, FIT-like toolset (Framework for Integrated Tests), the FATs can be executed continuously in the course of each iteration. Where automation is not possible, then these tests tend to build overhead, or technical debt, that must be executed at least at each system increment. Running the full set of feature regression tests, including all manual tests, is an integral part of system validation.

Caution should be taken here with identifying too many manual tests. To achieve the benefits of continuous integration in agile at all levels of the system and it sub systems, need to think about every way possible to automate testing. This will ensure that all testing and regression testing can be easily completed for each iteration and release without wasteful time delays. As teams transition to agile, need to think about the transition to continuous integration with automation testing and building a regression test suite.

The goal for the agile team is to ultimately have an environment where each and every change is submitted to the master repository and tested via the regression test suite. This will help the team achieve continuous integration. In addition, continuous integration is coupled with continuous V&V as the V&V activities are performed with each change submitted to the master repository to ensure all regulatory standards are adhered

to. Furthermore, this is coupled with continuous improvement as the team makes changes to the product, the team learns how to continuously improve the product and their process through inspection and adaption. Figure 12 below depicts how the team will continuously and iteratively develop the product with requirements, design, test, code many times within an iteration cycle. Note that the goal is to write automated tests before coding as the team transitions to Test Driven Development. The figure also depicts at the same time working towards continuous unit, integration, system and product testing. This shift will greatly help the team meet the goal of a PSI (potentially shippable increment) at the end of the iteration. The idea on continuous integration, V&V and improvement is the cornerstone of why agile is such a great methodology for high assurance environments. It breaks all of the regulatory requirements down to small activities so that the quality of those activities is more effectively met and in the end produces the highest quality product that has the highest value to the customer.

The concept and success of high quality with continuous iteration, integration and V&V is to do a little bit of everything all the time during product development. This gets away from the large bundling of the product for V&V and testing, as depicted in the V Model in Figure 4, that can be more prone to higher defect rates and actually less V&V rigor due to the large product bundle.

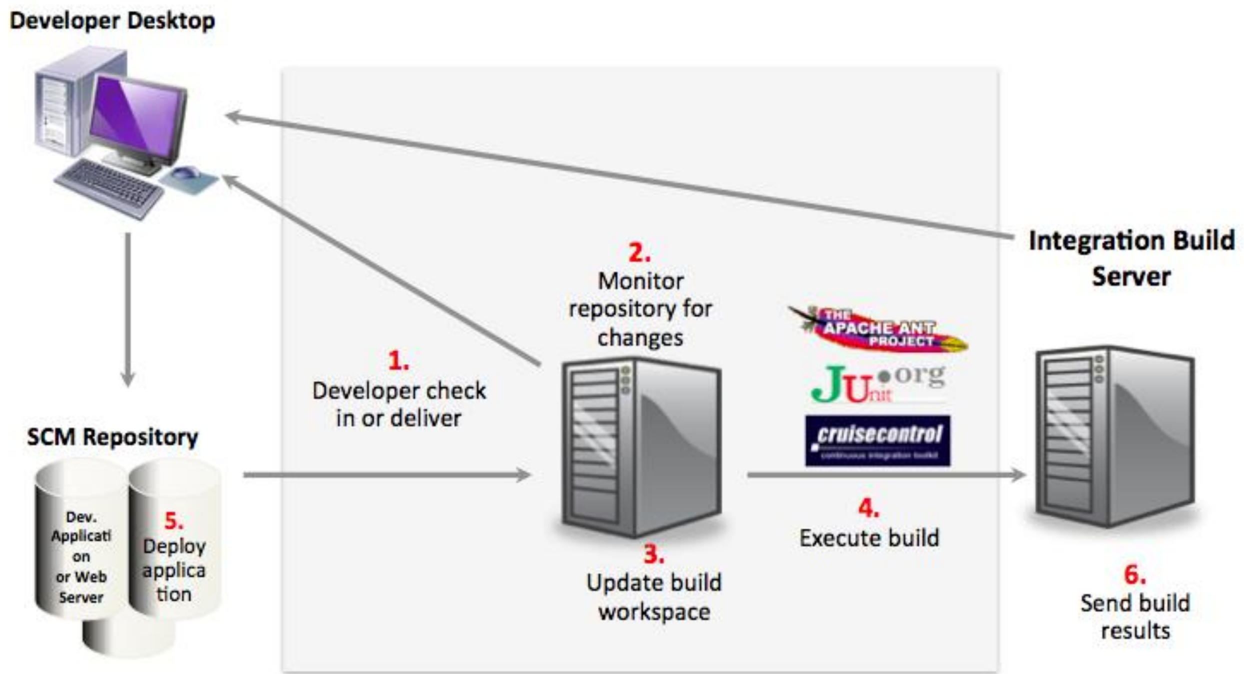


Figure 12 Continuous Iteration with Continuous Integration Flow

Testing Nonfunctional Requirements

In *Agile Software Requirements*, [Leffingwell 2011] I described nonfunctional requirements as the "URPS" part of our "FURPS" (Functionality, Usability, Reliability, Performance and Supportability) requirements categorization acronym and noted the following discriminators:

> **Usability** – Includes elements such as expected training times, task times, number of control activities required to accomplish a function, help systems, compliance with usability standards, usability safety features

Reliability – includes such things as availability, mean time between failures (MTBF), mean time to repair (MTTR), accuracy, precision, security, safety and override features

Performance – response time, throughput, capacity, scalability, degradation modes, resource utilization.

Supportability (Maintainability) – ability of the software to be easily modified to accommodate planned enhancements and repairs

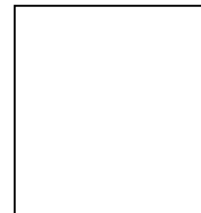
In addition, Design Constraints can also be particularly relevant in the development of high assurance systems. These can refer to items such as: *follow all internal processes per the companies Quality Management System,* and *code must adhere to IEC 60601-1 safety standards*.

No matter their nature or source, these requirements are just as critical as the functional requirements we have described in the PRD and user story-based SRS. If a system isn't reliable (becomes unavailable on occasion) or marketable (fails to meet our regulatory requirements) or isn't as accurate as the patient/user requires, then, Agile or not, we will fail just as badly as if we forgot some critical functional requirement.

As seen in the Figure 13 below, we modeled NFRs as constraints on new development, since many of these must be revisited at every increment to make sure the system — with all its new features — *still* meets this set of imposed quality requirements.

To assure that the system works as intended, most identified NFRs must typically be associated with some *System Qualities Test* used to validate that the system is still in compliance with that specific NFR. Testing these requirements is different from testing functional system requirements, for example:

- some can be tested by inspection only. Example: `all included software components must be validated`
- some must be tested with special harnessing or equipment, which may not be practical in each iteration. Example: `application pressure must be accurate to within +/50 millibars across the entire operating range`.
- some require continuous reasoning and analysis each time the system behavior changes (at each increment). Example: `adhere to IEC 60601-1 device safety standard`.



Because the testing of many NFRs is simply not automatable, some amount of manual NFR testing is left to the validation sprint. If the items can be automated, all the better, but either way, comprehensive regression testing (and documentation of results) of these system qualities is an integral part of the validation process.

Finalizing Documents

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

If an increment is intended for use in any actual user scenario — and after all, that is the intent — then the Device History records, including the PRD and SRS documents, tests and test results, etc., will need to be updated and finalized. This may include final content updates, exports as necessary, reformatting, and entering into the enterprise's document management system. There they can be logged and signed by the appropriate authorities and persisted for as long as that version of the system is subject to actual use. This documentation will serve as the objective evidence to satisfy the regulatory requirements.

Finalizing Traceability Matrices

In a like manner, traceability matrices must be updated, logged, signed and persisted in the device history files as well. This will include the traceability paths we have described above — including PRS to SRS, PRD to Feature Acceptance Test, User Story (SRS) to code, unit test and Story Acceptance Test, Nonfunctional Requirements to System Qualities Tests, and tracing tests that fail to defects — as well as any other such traceability relationships required by the companies Quality Management System. Having these traceability matrices is key in tracking objective evidence to meet regulatory requirements for verification and validation of requirements through the system from PRD to SRS to test that were executed for all requirements. The traceability depicted in the appendices supports the required traceability of the artifacts in the system that will serve as objective evidence.

Conclusion

Driven by quantified improvements in productivity, quality and employee morale, Agile software development methods are making their way into one of the last bastions of waterfall development, those industries developing high assurance software where the cost of defects and solution failures is unacceptably high.

We analyzed one industry example, looking at the regulations and guidance associated with the development of medical devices under the auspices of the US FDA. Perhaps surprisingly to some, we have confirmed that current regulatory advice that dates back more then a decade, does *not* mandate waterfall development. Rather, many of these guidance documents counsel the industry *away* from waterfall development and *towards* concurrent engineering. We also highlighted the mandated requirement for software verification and validation, and how in turn, effective V&V is dependent on maintaining correct and specific requirements documents, including software requirements specifications, product requirements documents, and traceability matrices.

For many, the state of the art for concurrent engineering in software development is based on Agile methods. In reviewing those methods, we put to rest another myth, that Agile development is loose, unstructured and is not necessarily based on a solid understanding of the exact system behavior a solution must deliver. To that end, we have suggested ways of doing Agile development in high assurance settings, focusing on short development increments with continuous verification, followed by validation iterations intended primarily to finalize solution testing and update the required documentation, including SRS, PRD and traceability matrices. We also shared examples — many provided by companies working in these environments today — as to how effective Agile tooling, from Rally Software and others, can support the documentation and verification and validation needs of high assurance development, while still enabling the team's day to day development activities to be based on proven Agile practices.

By illustrating these practical approaches to applying Agile methods in the high assurance context, we seek to help our industries deliver ever-higher quality safety, security and efficacy to our users. In turn, those enterprises with the courage to apply these methods can expect to gain the economic, enterprise and societal benefits available to those leaders who deliver the best products, and do so in the most efficient manner. This will lead high assurance companies to more competitive pricing and with decreased time to market for their products. Agile methods is the preferred methodology to achieve continuous integration that strives for zero-defect delivery of the highest quality products with the highest value to the customer. Automation and

tool usage is not a nice to have for high assurance companies, it is mandatory to achieve continuous integration as well as to satisfy the regulatory requirements for proper collection of objective evidence. This can be attained with the rigor and accountability required to meet regulatory standards in agile and with tooling coupled with the constant flow of communication within agile teams to be honest and transparent.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Bibliography

Abbot Laboratories and AgileTek. 2009. *Adopting Agile in an FDA regulated Environment.* Agile 2009 Conference Whitepaper. IEEE Computer Society, 978-0-7695-3768-9/09. Available online at http://www.computer.org/portal/web/csdl/doi/10.1109/AGILE.2009.50.

FDA documents available online at http://www.fda.gov/.

CFR - Code of Federal Regulations Title 21.

CFR 21 PART 820 – Quality System regulation

CFR 21 Subpart C – Design Controls. Subpart C, Sec. 820.30 – Design controls.

General Principles of Software Validation; Final Guidance for Industry and FDA Staff. 1997. Center for Devices and Radiological Health (CDRH).

Design Control Guidance for Medical Device Manufacturers. 1997. CDRH.

FDA Glossary of Computer Systems Software Development Terminology

Leffingwell, Dean. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Boston, MA: Addison-Wesley.

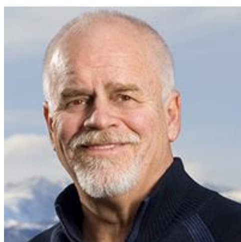
Leffingwell, Dean. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston, MA: Addison-Wesley.

Leffingwell, Dean, and Don Widrig. 2003. *Managing Software Requirements, Second Edition: A Use Case Approach*. Boston, MA: Addison-Wesley.

Leffingwell, Dean, and Don Widrig. 2000. *Managing Software Requirements: A Unified Approach*. Boston, MA: Addison-Wesley.

Reinertsen, Donald G. 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development*. Redondo Beach, CA: Celeritas Publishing.

About the Author



Dean Leffingwell is a renowned entrepreneur, software executive, consultant and author. Mr. Leffingwell founded or cofounded a number of companies including medical equipment company RELA/Colorado Medtech, software tools maker Requisite, Inc., now part of IBM's Rational Division, and most recently, Scaled Agile, Inc. Mr. Leffingwell formerly served as chief methodologist to Rally Software and, before that, as Vice President of Rational Software, now IBM's Rational Division, where his responsibilities included the Rational Unified Process.

Mr. Leffingwell has been a student, coach and author of contemporary software engineering and management practices throughout his career. His most recent project is the ScaledAgileFramework.com. His books include *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise* (Addison-Wesley 2011*)* and *Scaling Software Agility: Best Practices for Large Enterprises* (Addison-Wesley 2007). He is also lead author of the texts *Managing Software Requirements: First and Second Editions*, which have been translated into five languages.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

About the Primary Contributor

Craig Langenfeld is a Product Ambassador for Rally Software. Since 2000, Craig has held such titles as developer, project manager, instructor, ScrumMaster, tool expert, and friend to those who are driving organizational change and seeking better ways to deliver software. Recently he has focused his efforts on building a community and sharing best practices for Agile software development within high assurance industries. He is also passionate about scaling Agile practices and tooling within large enterprises.

Yvonne Kish is an Agile Coach with Rally Software in Boulder, CO. Yvonne started her career in software engineering as a software developer. She transitioned to software roles where she could interact more with the software development process as a whole. She has held positions as a software configuration manager, software quality assurance engineer and manager, process improvement consultant, and Agile coach. She has experience in many high assurance type areas including the DoD, FAA, and FDA.

Yvonne is passionate about helping companies use Agile to ultimately produce higher quality products that successfully meet customer regulatory requirements.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Appendix

This appendix highlights examples of using Rally Software's Agile Lifecycle Management platform, often in conjunction with other products such as Subversion and HP's Quality Center, to support the documentation, verification, validation, and traceability needs of high assurance software development, while still driving fundamental Agile development practices for the teams day to day development activities. A special thanks to Craig Langenfeld of Rally Software for developing many of these examples, and collecting others from practitioners doing this actual work in the field today.

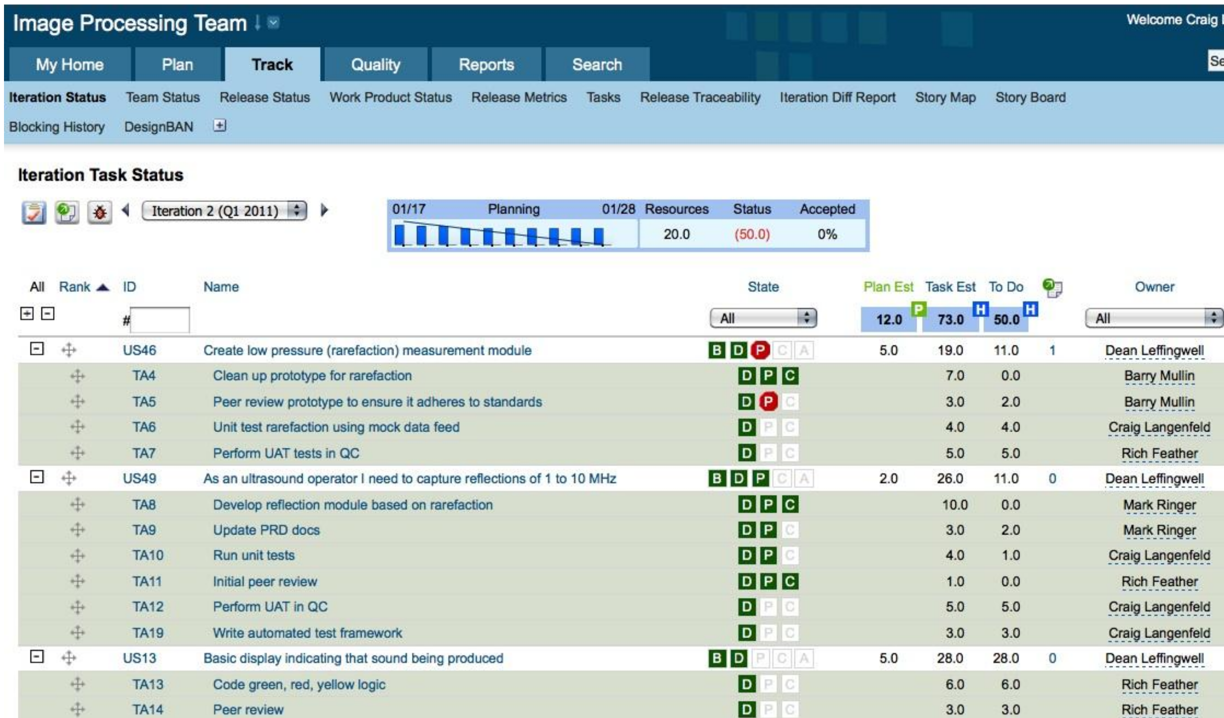
Figure A 1.1 – Feature set represented in Rally

ID	Name	Project	Release	Iteration	State	Plan Est	Owner	Tags
US96	PRD: EPAT (Extracorporeal Pulse Activation Therapy) Device Enhancement	Ultrasound Device 2000X			B D P C A	11.0	Craig Langenfeld	PRD SRS Theme
US97	SRS: Pulse wave frequency is adjustable from 1-21 hertz	Ultrasound Device 2000X			B D P C A		Craig Langenfeld	SRS
US93	SRS: Acoustic energy may be concentrated to a focal area of 2- 8 mm in diameter	Image Processing Team	2011 Q2 Release	Iteration 5 (Q2 2011)	B D P C A	3.0	Craig Langenfeld	SRS
US92	SRS: Pulse amplitude is adjustable from 1-5 bar	Ultrasound Device 2000X			B D P C A	8.0	Craig Langenfeld	SRS
US95	As an operator deliver 5 bar with no further effect	Display Team			B D P C A	3.0	Craig Langenfeld	Functional
US94	As an operator pulse amplitude setting display in .1 bar increments	Image Processing Team			B D P C A	5.0	Craig Langenfeld	Functional

In the screen shot above we show a PRD, in this case "EPAT Device Enhancement", represented by a parent user story in Rally. The indented rows beneath the parent story represent child stories or what Dean Leffingwell describes as the SRS level in the hierarchy. Rally offers a robust user story hierarchy that decomposes n-levels deep to easily support the requirements structure that is referred to in this writing. Traceability in Rally provides ability to build relationships between stories in the hierarchy and rollup data from child stories and development tasks to parent stories. Information such as state progress and plan estimates become visible across the hierarchy. Additionally, Rally provides strong reporting capabilities across multi level hierarchy of decomposed stories.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Figure A 1.2 – Rally Iteration Status View



Rally's Iteration Status view, shown above, guides teams to meet their commitments by providing clear visibility to define|build|test activities throughout the Iteration. This example shows a team that has planned three user stories for Iteration 2. The team defined their work by decomposing the stories into tasks and is able to track their effort against those tasks.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Figure A 1.3 – User Story detail view highlights traceable relationships

In this whitepaper, we have highlighted the importance of User Story to Acceptance Test Case traceability in high assurance environments, but we have not yet highlighted the importance of other artifacts like defects, changesets, and revisions. Rally's unique User Story detail view provides clear visibility to how many defects are associated to this story, what the test coverage is, what changesets are associated, and an audit trail for revision information.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Figure A 1.4 – User Story to Code Traceability

ID	Name	State					
US46	Create low pressure (rarefaction) measurement module	B D P C A	5.0	19.0	11.0	1	Dean Leffingwell
TA4	Code prototype for rarefaction	D P C		7.0	0.0		Barry Mullin
TA5	Peer review prototype to ensure it adheres to standards	D P C		3.0	2.0		Barry Mullin
TA6	Unit test rarefaction using mock data feed	D P C		4.0	4.0		Craig Langenfeld
TA7	Perform UAT tests in QC	D P C		5.0	5.0		Rich Feather

Above is a simple example of using user stories and tasks to create a traceability path. In this case we show a single User Story that has four tasks, including "TA4 Code prototype for rarefaction". Having a tight association between a user story and the actual coding tasks that implement it is critical in high assurance environments. The status of the User Story is directly rolled up from the status of the connected tasks ensuring that changes to tasks trace to the User Story in real-time.

Healthcare Data
Image Processing Team
Project Program Setup
Welcome Craig Langenfeld | Help | Logout
My Home Plan Track Quality Reports Search
Iteration Status Team Status Release Status Work Product Status Release Metrics Tasks Release Traceability **Iteration Diff Report** Story Map
Story Board Blocking History DesignBAN Build Traceability

Select
Iteration: Iteration 2 (Q1 2011)
Sort By Revision File

ID	Name	Revision	Author	File	Changed
TA4	Clean up prototype for rarefaction	11 [TA4 Completed some comments here]		src/org/rallydev/Sample.java	prev, all
TA4	Clean up prototype for rarefaction	10 [TA4]		src/org/rallydev/Sample.java	prev
TA4	Clean up prototype for rarefaction	9 [TA4 In-Progress Barry]		src/org/rallydev/Sample.java	
TA8	Develop reflection module based on rarefaction	12 [TA8 In-Progress added code to reflection module]		src/org/rallydev/Sample.java	, all , all , all

In addition, Rally's integration with SVN automates the traceability between a code revision and a Rally task. In the example above, a custom Iteration Diff Report has been created to report on all files that were changed during an Iteration. The Iteration Diff Report is a Rally App example that was created using Rally's App Development Platform.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Figure A 1.5 – User Story to Acceptance Tests in Rally

ID	Name		Verdict	Build	Date	
US13	Basic display indicating that sound being produced	B D P C A	Pass: 3/4	Fast_Run_3-25_1-44-38	03/24/2011	1
TC22	10 MHz of sound being produced for US13		Pass	Fast_Run_3-25_1-44-24	03/24/2011	
TC23	Display green light if sound is produced for US13		Fail	Fast_Run_3-25_1-44-11	03/24/2011	
TC24	Red light display if no sound is produced for US13		Pass	Fast_Run_3-25_1-44-38	03/24/2011	
TC25	Yellow light displayed if sound is over 10 MHz for US13		Pass	Fast_Run_3-25_1-42-44	03/24/2011	

User Story Acceptance Tests (SAT) are functional tests intended to assure that the implementation of each new user story delivers the expected behavior. In a regulated environment it is often referred to as verification testing. Above is a screen capture from Rally showing four verification test cases, and their results, associated to a user story. Having clear visibility from the user story to the verification test case is critical for Agile teams as they work through an Iteration.

User Story US13: Basic display indicating that sound being produc...

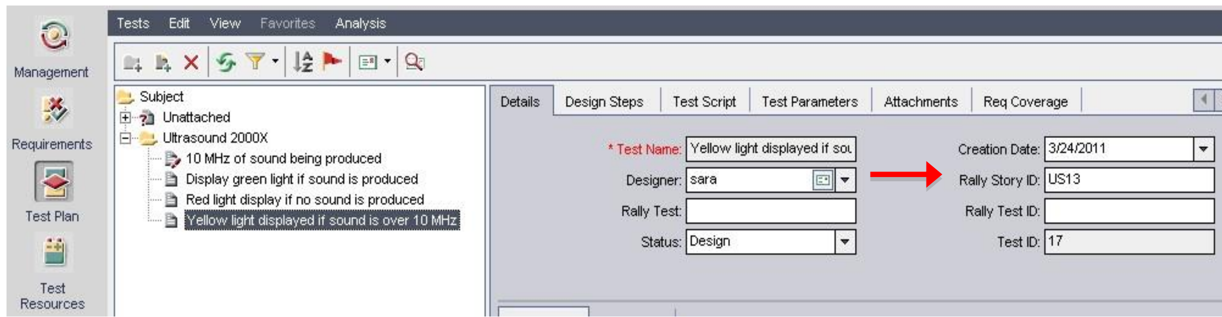
- Details for US13
- Children (0)
- Predecessors (0)
- Successors (0)
- Tasks (6)
- Defects (1)
- Test Cases (4)
- Test Run (4)
- Chart
- Changesets (0)
- Discussion (0)
- Attachments (0)
- Revisions (21)

Test Cases

ID	Name	Work Product	Type	Priority	Owner	Method	Last Verdict	Last Build	Last Run
TC18	10 MHz of sound being produced	US13: Basic display indicating that sound being produced	Verification			Manual	Pass	Run_3-24_22-32-14	03/24/2011
TC19	Display green light if sound is produced	US13: Basic display indicating that sound being produced	Verification			Manual	Fail	Fast_Run_3-24_22-41-1	03/24/2011
TC20	Red light display if no sound is produced	US13: Basic display indicating that sound being produced	Verification			Manual	Pass	Fast_Run_3-24_22-41-5	03/24/2011
TC21	Yellow light displayed if sound is over 10 MHz	US13: Basic display indicating that sound being produced	Verification			Manual	Pass	Fast_Run_3-24_22-41-10	03/24/2011

The screen capture above shows another view of the User Story Acceptance Tests in context of the User Story they trace to.

Figure A 1.6 – User Story to Acceptance Test Traceability (Rally and Quality Center)



Above screenshot shows a test plan view in QC, which displays four User Story Acceptance Test Cases that have been defined for the Ultrasound 2000X machine. In this example, the detail of the highlighted test case has a Rally Story ID field. Populating this field and running the Rally/QC connector allows us to synchronize this set of test cases with Rally.

US13	Basic display indicating that sound being produced	B D P C A	Pass: 3/4	Fast_Run_3-24_22-41-10	03/24/2011	1	Dean Leffingwell
TC18	10 MHz of sound being produced for US13		Pass	Run_3-24_22-32-14	03/24/2011		
TC19	Display green light if sound is produced for US13		Fail	Fast_Run_3-24_22-41-1	03/24/2011		
TC20	Red light display if no sound is produced for US13		Pass	Fast_Run_3-24_22-41-5	03/24/2011		
TC21	Yellow light displayed if sound is over 10 MHz for US13		Pass	Fast_Run_3-24_22-41-10	03/24/2011		

The screen capture above displays the same four test cases that were created and ran in QC as Rally Test Cases associated with a Rally User Story. The test cases were created in Rally by the Rally/QC connector and their results will be updated each time they are run from QC. Having this tight association between User Stories and Test Cases is required for visibility, traceability and assurance.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Figure A 1.7 – Validation Requirements Traceability Matrix Example

Traceability Matrix Page Tools

Choose a Product Requirement

US96 - PRD: EPAT (Extracorporeal Pulse Activation Therapy) Device Enhancement

User Story	Test Case ID	Test Case Name	Type	Source	Risk	Owner	Last Run	Last Verdict	Design Output
US96-PRD: EPAT (Extracorporeal Pulse Activation Therapy) Device Enhancement	TC35	Create pulse amplitude of 1 bar	Validation		Medium	Craig Langenfeld	...	...	...
US96-PRD: EPAT (Extracorporeal Pulse Activation Therapy) Device Enhancement	TC36	Indicator is green between 1 and 5 bar	Validation		Medium	Craig Langenfeld	...	...	...

SRS Traceability Table

User Story	Test Case ID	Test Case Name	Type	Source	Risk	Owner	Last Run	Last Verdict	Design Output
US93-SRS: Acoustic energy may be concentrated to a focal area of 2- 8 mm in diameter	...	...	...		...	...	...	...	...
US92-SRS: Pulse amplitude is adjustable from 1-5 bar	TC29	Pulse equal 1 bar	Verification		Low	Craig Langenfeld	05/17/2011	Pass	...
US92-SRS: Pulse amplitude is adjustable from 1-5 bar	TC30	Pulse increase from 1 to 2	Verification		Low	Craig Langenfeld	05/17/2011	Pass	PRD Trace Matrix.tiff
US92-SRS: Pulse amplitude is adjustable from 1-5 bar	TC31	Pulse increase from 1 to 5	Verification		Low	Craig Langenfeld	05/17/2011	Fail	...
US97-SRS: Pulse wave frequency is adjustable from 1-21 hertz	TC32	Green light indicator at 5	Verification		Low	Craig Langenfeld	05/17/2011	Pass	...
US97-SRS: Pulse wave frequency is adjustable from 1-21 hertz	TC33	Yellow light indicator at 6	Verification		Medium	Craig Langenfeld	05/17/2011	Pass	...
US97-SRS: Pulse wave frequency is adjustable from 1-21 hertz	TC34	Red light indicator at 7	Verification		High	Craig Langenfeld	05/17/2011	Pass	...

Guidances like, *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*, strongly suggest that medical device manufacturers should produce traceability matrices as a validation design output artifact. As Dean Leffingwell explains, traceability matrices provide evidence to stakeholders that we traced the intent of the system, to the design, to the quality of the system. The figure above shows how Rally can be extended to create a custom traceability matrix displaying the traceability from PRD to SRS and even further into test cases and test results. Automating the production of artifacts like the traceability matrix is key for Agile teams which work in shorter release cycles.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation

Figure A 1.8 – Product Requirements Document Export from Rally

Ultrasound Device 2000X

Welcome Craig Langenfeld | Help | Log

My Home | Plan | Track | Quality | Reports | Search

Search

System Requirement Validation Document

Page Tools

Choose a System Requirement

Product Requirement

Product Requirement:	US1 PRD:Transmit high-frequency sound so that it pulses into the patient
System Requirement:	US7 SRS:Produce sound using piezoelectric transducer
Acceptance Date:	Not Accepted Yet

System Requirement Description

Specification:	US7 SRS:Produce sound using piezoelectric transducer
Description:	As a user I want to produce sound using a piezoelectric transducer. **System Specs** The system shall convert electricity into vibrations. The system shall produce a wavelength that is twice the size of the element's thickness. **Non-Functional** The system must perform at X
Functional Stories:	US12 Establish development environment (1 Test Cases) US13 Basic display indicating that sound being produced (4 Test Cases) US86 The system shall convert electricity into vibrations. (0 Test Cases) US87 The system shall produce a wavelength that is twice the size of the element's thickness. (0 Test Cases) US88 The system must perform at X (0 Test Cases)

Test Case

Test Case:	TC17 Establish development environment
Story:	US12 Establish development environment
Test Case Description:	This is a test description.
Last Run:	08 Apr 2011

As Dean describes above, in high assurance software development we will need to format our user stories, test cases and results, and defects in a way that they can be easily exported to represent that the intended feature was built to its specification. Above is an example of a Product Requirements Document that was created using Rally's App SDK and can be added to Rally via the App Catalog. Generated documents can be exported for formal signoff and storage in a document repository. Having an automated mechanism for creating artifacts like a PRD are key in a fast moving Agile Lifecycle.

© 2011-2014 Leffingwell, LLC. and Rally Software Development Corporation