Technical Information Report

AAMI TIR45: 2023

Guidance on the use of AGILE practices in the development of medical device software

AAMI
Advancing Safety in Health Technology

Guidance on the use of AGILE practices in the development of medical device software

Approved 15 March 2023 by
AAMI

Abstract: Over the past several years, **AGILE** software development has become an accepted method for developing software products. There have been questions from both manufacturers and regulators as to whether (or which) **AGILE** practices are appropriate for developing medical device software. Enough medical device manufacturers have implemented **AGILE** practices in their software development so that answers to these questions can be documented. Having clear guidance of which practices have been found to be appropriate will be very useful for all developers of medical device software. This TIR will provide recommendations for complying with international standards and U.S. Food and Drug Administration (FDA) guidance documents when using **AGILE** practices to develop medical device software.

Keywords: **AGILE**, software

AAMI Technical Information Report

A technical information report (TIR) is a publication of the Association for the Advancement of Medical Instrumentation (AAMI) Standards Board that addresses a particular aspect of medical technology.

Although the material presented in a TIR may need further evaluation by experts, releasing the information is valuable because the industry and the professions have an immediate need for it.

A TIR differs markedly from a standard or recommended practice, and readers should understand the differences between these documents.

Standards and recommended practices are subject to a formal process of committee approval, public review, and resolution of all comments. This process of consensus is supervised by the AAMI Standards Board and, in the case of American National Standards, by the American National Standards Institute.

A TIR is not subject to the same formal approval process as a standard. However, a TIR is approved for distribution by a technical committee and the AAMI Standards Board.

Another difference is that, although both standards and TIRs are periodically reviewed, a standard must be acted on—reaffirmed, revised, or withdrawn—and the action formally approved usually every five years but at least every 10 years. For a TIR, AAMI consults with a technical committee about five years after the publication date (and periodically thereafter) for guidance on whether the document is still useful—that is, to check that the information is relevant or of historical value. If the information is not useful, the TIR is removed from circulation.

A TIR may be developed because it is more responsive to underlying safety or performance issues than a standard or recommended practice, or because achieving consensus is extremely difficult or unlikely. Unlike a standard, a TIR permits the inclusion of differing viewpoints on technical issues.

CAUTION NOTICE: This AAMI TIR may be revised or withdrawn at any time. Because it addresses a rapidly evolving field or technology, readers are cautioned to ensure that they have also considered information that may be more recent than this document.

All standards, recommended practices, technical information reports, and other types of technical documents developed by AAMI are *voluntary*, and their application is solely within the discretion and professional judgment of the user of the document. Occasionally, voluntary technical documents are adopted by government regulatory agencies or procurement authorities, in which case the adopting agency is responsible for enforcement of its rules and regulations.

Comments on this technical information report are invited and should be sent to AAMI, Attn: Standards Department, 901 N. Glebe Road, Suite 300, Arlington, VA 22203.

Published by

AAMI
901 N Glebe Rd, Suite 300
Arlington, VA 22203
www.aami.org

Printed in the United States of America

ISBN 978-1-57020-868-3

Contents

Page

Figures

Tables

Committee representation

Association for the Advancement of Medical Instrumentation

Medical Device Software Committee

This AAMI Technical Information Report was developed by the AAMI Agile Software Task Group under the auspices of the AAMI Medical Device Software Committee. Approval of the Technical Information Report does not necessarily mean that all members voted for its approval.

At the time this document was published, the **AAMI Medical Device Software Working Group** had the following members:

Cochairs:
Michelle Jump
Lisa Simone

Members:
Pat Baird, Philips
Kimberly Colasanti, Baxter Healthcare Corporation
Sandra Robyn Curtis, ResMed Inc.
William K. Day, Near Future Corp
Richard W. De La Cruz, Silver Lake Group Inc
Sherman Eagles, SoftwareCPR
Christine M. Flahive, Chris Flahive Associates
Kenneth J. Fuchs, Draeger Medical Systems Inc.
Regis George, Eli Lilly & Company
Kerry A. Griffin, Stryker
Jessica Hagg, Smiths Medical
Jeffrey A. Hallett, (Independent Expert)
James Hamlyn, Skin Analytics Ltd.
Rickey L. Hampton, Mass General Brigham
David Hendrickson, Spacelabs Healthcare
Lezlie L. Hynes, NAMSA
Jeremy J. Jensen, Boston Scientific Corporation
Michelle Jump, MedSec
Patricia Krantz-Zuppan, Medtronic Inc
Robert Kruth, Johnson & Johnson
Sucheth Koppa, Roche Sequencing Solutions, Inc
Bo (Tyler) Li, Telesair Inc
Yimin Li, Astellas Pharma Inc.
Mulugeta Mideksa (Independent Expert)
Curtis Morgan, 3M Health Care
D. Scott Mueller, AbbVie
Vidya Murthy, MedCrypt
Susumu Nozawa, Siemens Healthineers
Horst Pichler, Fresenius Medical Care
Melissa Purpura, STERIS Corporation
Hugo Felix Rodriguez, Pfizer
Ann Rossi, Vicarious Surgical, Inc.
Daniel Rubery, Fresenius Medical Care
Ron Seese, Arthrex Inc
Ray P. Silkaitis. Amgen Inc
Lisa Simone, FDA/CDRH
Jason R. Smith, UL LLC
Ivo Toptchev, ICU Medical Inc
Swati Totawat, Sanofi
Gary Turner, SunTech Medical Inc

Loren Walkington, Cordis US Corp
Mark Walker, Walker Validation and Compliance

Alternates:
Felix Alejandro, Cordis US Corp
Ron Baerg, SoftwareCPR
Wei Bao, Personal Genome Diagnostics
Christopher J. Brown, FDA/CDRH
Uma Chandrashekhar, Alcon Laboratories Inc
Brian Conn, Medtronic Inc Campus
Divya Enika-Patel, Johnson & Johnson
Edwin Heierman III, Abbott Laboratories
Magnus Miller-Wilson, Boston Scientific Corporation
Ian Mitchell, Spacelabs Healthcare
Mark Rohlwing, ICU Medical Inc
Diane Sheffer, STERIS Corporation
Monica R. Singireddy, ResMed Inc.
Fei Wang, Fresenius Medical Care
Nicola Zaccheddu, Philips
Nicole Zuk, Siemens Healthineers

Liasions:
Keith Anderson, Smiths Medical
Igor Bendersky, AbbVie
Greg Bevan, Abbott Laboratories
Ying Chen, DexCom Inc
Shradha Chopra, DexCom Inc
Katie Chowdhury, AbbVie
Tatiana Correia, 3M Health Care
Martin Joseph Crnkovich, Fresenius Medical Care
Jean R. Desire, ICU Medical Inc
Brian J. Fitzgerald, FDA/CDRH
Anthony Hardcastle, DexCom Inc
James P. Hempel, Medtronic Inc
Kartik Iyer, Siemens Healthineers
Michael B. Jaffe, Philips
Ujjwal Jain, Illumina Incorporated
Joshua Kim, Baxter Healthcare Corporation
Jyh-Shyan Lin, DexCom Inc
Don McAndrews, Philips
David D. Nelson, Boston Scientific Corporation
David G. Osborn, Philips
Mike Owen, Fresenius Medical Care
Xenophon Papademetris
Milind Patel, AbbVie
Robert Z. Phillips, Siemens Healthineers
Sheila Ramerman, DexCom Inc
Peter Rech, SoftwareCPR
William Roeca, Becton Dickinson & Company
Xianyu Shea, Stryker
Scott Thiel, Hologic Inc
Kelly Weyrauch, Agile Quality Systems LLC
Vivian Wu, DexCom Inc
Charles Zinsmeyer, 3M Health Care
Ling Zou, Becton Dickinson & Company

NOTE Participation by federal agency representatives in the development of this technical information report does not constitute endorsement by the federal government or any of its agencies.

At the time this document was published, the **AAMI Revision Team Members** were the following.

Project Leads: Pat Baird
Patricia Krantz-Zuppan

Members: Richard W. De La Cruz, Silver Lake Group Inc
Lezlie L. Hynes, NAMSA
Michael Iglesia, Roche
Jeremy J. Jensen, Boston Scientific Corporation
Scott Mueller, Abbvie
Sandra Robyn Curtis, ResMed Inc.
William Sammons, Hillrom
Lisa Simone, FDA/CDRH
Kelly Weyrauch, Agile Quality Systems LLC
Nicola Zaccheddu, Philips

NOTE Participation by federal agency representatives in the development of this technical information report does not constitute endorsement by the federal government or any of its agencies.

NOTE Participation by federal agency representatives in the development of this technical information report does not constitute endorsement by the federal government or any of its agencies.

Foreword

Over the past several years, **AGILE** software development has become an accepted method for developing software products. There have been questions from both manufacturers and regulators as to whether (or which) **AGILE** practices are appropriate for developing medical device software. Enough medical device manufacturers have implemented **AGILE** practices in their software development so that answers to these questions can be documented. Having clear guidance of which practices have been found to be appropriate will be very useful for all developers of medical device software.

This TIR will provide recommendations for complying with international standards and U.S. Food and Drug Administration (FDA) regulations and guidance documents when using **AGILE** practices to develop medical device software.

The concepts incorporated herein are not inflexible or static. They are reviewed periodically to assimilate new data and advances in technology.

The following verbal forms are used within AAMI documents to distinguish requirements from other types of provisions in the document:

- "shall" and "shall not" are used to express requirements;
- "should" and "should not" are used to express recommendations;
- "may" and "may not" are used to express permission;
- "can" and "cannot" are used as statements of possibility or capability;
- "might" and "might not" are used to express possibility;
- "must" is used for external constraints or obligations defined outside the document; "must" is not an alternative for "shall."

Suggestions for improving this recommended practice are invited. Comments and suggested revisions should be sent to Standards, AAMI, 901 N Glebe Road, Suite 300, Arlington, VA 22203 or standards@aami.org.

NOTE This foreword does not contain provisions of the AAMI TIR45, *Guidance on the use of* **AGILE** *practices in the development of medical device software* (AAMI TIR45:2023), but it does provide important information about the development and intended use of the document.

Introduction

AGILE software development (hereafter referred to simply as "**AGILE**") has been evolving for many years. **AGILE** began as a niche concept being used in small pockets of the software industry and has since grown to be well established in many different software development contexts. As it has grown, it has been adapted to fit the unique needs of a specific context. For **AGILE** to be established in the medical device software industry, guidance is needed to adapt it to fit that unique context. This TIR fulfills that need.

Why read this TIR?

AGILE was developed in response to quality and efficiency concerns posed by existing methods of software development. It can bring benefits that are valuable to the medical device software world, including the following:

- Continuous focus on safety, risk management, and delivering customer value through **BACKLOG** prioritization, planning practices, and customer feedback;
- Continuous assessment of quality through continuous integration and testing;
- Continuous improvement of the software development process through **RETROSPECTIVES** and team accountability;
- Continuous focus on "getting to DONE" and satisfying quality management stakeholders through the regular completion of activities and deliverables.

AGILE can bring value to medical device software.

There are concerns about **AGILE**'s compatibility with the regulated world of medical device software development. For example, the **AGILE** Manifesto has value statements that seem contrary to the values of a quality management system; and because **AGILE** initially grew from the information-technology space where human safety and risk management may not of primary importance, there is concern that **AGILE** lacks the proper controls for producing safety-critical software.

Fortunately, **AGILE**'s fundamental nature is to be adaptable to the context in which it is applied, allowing for **AGILE** principles and practices to be applied in ways that are compatible with the needs of the safety-critical, medical device software world.

***AGILE** can be adapted to the unique needs of medical device software.*

This TIR will examine **AGILE**'s goals, values, principles, and practices, and provide guidance on how to apply **AGILE** to medical device software development. It will

- provide motivation for the use of **AGILE;**
- clarify misconceptions about the suitability of **AGILE;** and
- provide direction on the application of **AGILE** to meet quality system requirements.

Following the guidance provided by this TIR can help medical device software manufacturers obtain the benefits provided by **AGILE** and satisfy regulatory requirements and expectations.

Initial recommendations

This TIR provides recommendations for ways to effectively apply **AGILE** to medical device software. Here are some of the initial recommendations that are explained further later.

AGILE is driven by the value statements written in the *Manifesto for **AGILE** Software Development*. These value statements can seem to be contradictory to the values of the regulated world of medical device software, but they need not be interpreted that way. Instead, they can be aligned to enhance the effectiveness of the quality management system.

*Apply the values of **AGILE** in a way that enhances a robust quality management system.*

AGILE emphasizes the need for the team to own its practices, inspect them, adapt them, and optimize them to their context. Regulatory requirements emphasize the need to establish a robust quality management system. Within the context of an established quality management system, **AGILE** practices can be applied without disrupting the quality system and without raising undue concern among regulators.

*Apply the practices of **AGILE** within the context of an established quality management system.*

AGILE embraces a highly **INCREMENTAL**/**EVOLUTIONARY** life cycle for software development. Although regulations and standards do not mandate a particular life cycle model, if stakeholders have expectations for linear life cycle models, an **INCREMENTAL**/**EVOLUTIONARY** life cycle might bring challenges.

*Set the correct expectations by defining the **SOFTWARE DEVELOPMENT LIFE CYCLE MODEL**. Demonstrate how an **INCREMENTAL/EVOLUTIONARY** life cycle satisfies regulatory requirements.*

As part of its **INCREMENTAL**/**EVOLUTIONARY** life cycle, **AGILE** emphasizes the ability to respond quickly to change. Because rapid change can increase risks to product quality, effective change management systems are essential to align the desire to change quickly and the need to manage risk.

Establish robust change management systems to manage changes and mitigate risks associated with rapid change.

Guidance on the use of AGILE practices in the development of medical device software

1 Scope

1.1 Inclusions

This Technical Information Report (TIR) provides perspectives on the application of **AGILE** during medical device software development. It relates them to the following existing standards, regulations, and guidance:

- ISO 13485:2016, *Quality management systems—Requirements for regulatory purposes;*
- IEC 62304:2006+AMD1:2015, *Medical device software–Software life cycle processes;*
- ISO 14971:2019, *Medical devices—Application of risk management to medical devices;*
- FDA *Code of Federal Regulations* (CFR), Title 21, Part 820.30, Quality System Regulation: Design Controls;
- FDA *Guidance for the content of premarket submissions for software contained in medical devices;*
- FDA *General principles of software validation; Final guidance for industry and FDA staff.*

Although this TIR does not provide a particular perspective for IEC TR 80002-1 (*Guidance on the application of ISO 14971 for medical device software*), the pertinent aspects of software risk management for medical devices were integrated throughout this TIR.

The following groups are the intended audience for this TIR:

- Medical device manufacturers who are planning to use **AGILE** techniques;
- Manufacturers who are currently practicing **AGILE** and are entering the regulated medical device space;
- Software development teams, including software test and quality groups;
- Software definers, including marketing, sales, and other representatives of the customer;
- Senior management, project managers, quality managers;
- Quality systems and regulatory affairs personnel;
- Internal and external auditors;
- Regulating bodies, agencies, and organizations responsible for overseeing the safety and effectiveness of medical devices.

1.2 Exclusions

This TIR is not intended to be used as an educational tool or tutorial for the following:

- **AGILE** development practice;

— Quality system regulations.

This TIR should be regarded as a reference and as a guidance intended to provide recommendations for complying with international standards and FDA guidance documents when using **AGILE** practices in the development of medical device software. This TIR is not intended to be a prescription for a specific situation or method.

1.3 Organization: Navigating this document

This TIR is organized into three main sections:

1) Setting the stage (Clause 4);

2) Aligning on concepts (Clause 5);

3) Aligning on practices (Clause 6).

Clause 4 provides background information necessary to understand the context of this TIR.

Subclause 4.1 describes the **AGILE** perspective, explaining the goals, values, principles, and practices that define **AGILE** software development. If you are new to the **AGILE** world, this section is a good place to start.

Subclause 4.2 describes the regulatory perspective: the goals, values, principles, and practices that define the regulated world of medical device software development. If you are new to the regulatory world, this section is a good place to start.

If you already have a working knowledge of both worlds, you could skip subsection 4.1 and subsection 4.2 and refer to them as necessary when reading other sections of this TIR. Neither of these subsections provides a complete, detailed description of these two perspectives; they provide only enough information to support the context of the rest of the TIR. Additional references are provided for readers who want more information.

Subclause 4.3 addresses some of the most important topics in aligning the **AGILE** and regulatory perspectives. It compares and contrasts the things that **AGILE** values with things that the regulatory perspective values. **AGILE** values are defined in the *Manifesto for **AGILE** Software Development* with statements that can be read as contrary to regulatory values. This subsection provides recommendations on how to align these different values in a supportive way. If the high-level goals and values of either perspective are a source of concern for your organization, this section might be a good place to start.

Clause 5 compares and contrasts some of the high-level concepts that define the **AGILE** and regulatory perspectives. **AGILE** provides many detailed practices but emphasizes the need to adapt them to fit the needs of a particular context, so it is important to understand the principles of **AGILE** when adapting the practices. Regulations and standards provide broad guidance on the requirements for a quality management system and require manufacturers to provide the details that describe their process, so it is important to understand regulatory principles when defining a quality management system. Many principles from these different perspectives align very well, whereas others might provide some challenges. This section provides recommendations on how to align these different principles in a supportive way. If you want to address foundational issues of principles and concepts before getting into details of implementation, this section would be a good place to start.

Clause 6 addresses many details of implementing **AGILE** in a regulated environment and provides many recommendations and considerations. The section is broken into large topic groups. If you want to get into the details of the specific implementation of a process or practice, this section would be a good place to start.

Most sections highlight an important point relevant to that section's topic.

Highlights look like this.

The text around these highlights provides more detail to support it.

It is not necessary to read this TIR in order from front to back. There is some repeated information in the various sections to give information needed to understand a topic, and there are cross-references in many sections to point to more information to understand a topic. Read the sections in the order that makes sense to you.

The committee realizes that this TIR covers a wide variety of agile-related topics, and some readers may be interested in just a subset of the topics. For example, some readers may be interested in the day-to-day management of an agile team, other readers may be more interested in how to ensure the activities are compliant with regulations and standards. Annex B has been developed to help readers quickly find guidance regarding agile-related activities that they are interested in.

2 Normative references

U. S. FDA Medical Device Quality System Regulation:

Code of Federal Regulations (CFR), Title 21, Part 820, Quality System Regulation

U. S. FDA Guidance:

General principles of software validation; Final guidance for industry and FDA staff, 2002

Guidance, *Guidance for the content of premarket submissions for software contained in medical devices,2005*

Off-the-shelf software use in medical devices; Guidance for industry and FDA staff, 2019

Applying human factors and usability engineering to medical devices, Final guidance for industry and FDA staff 2016

Design Control Guidance for Medical Device Manufacturers

1997 EU Medical Device Directives:

European Union Medical Device Regulation (2017/745) Medical Device

Standards

IEC 62304:2006+amd1:2015, *Medical device software—Software life cycle processes*

ISO 14971: 2019, *Medical devices—Application of risk management to medical devices*

ISO 13485: 2016, *Quality management systems—Requirements for regulatory purposes*

Technical Reports

IEC/TR 80002-1:2009, Technical Report—*Medical device software—Part 1: Guidance on the application of ISO 14971 to medical device software*

AAMI TIR57:2016, Technical Report - *Principles for medical device security – Risk management*

AGILE References

Martin **RC.** ***AGILE*** *Software Development: Principles, Patterns, and Practices*. Upper Saddle River (NW): Prentice Hall, 2003.

Cohn M. *User Stories Applied for* ***AGILE*** *Software Development*. Reading (MA): Addison-Wesley, 2004.

Cohn M. **AGILE** *Estimating and Planning*. Upper Saddle River (NJ): Prentice Hall, 2006.

Beck K. *Test-Driven Development by Example*. Reading (MA): Addison-Wesley, 2003.

Poppendieck M, and Poppendieck T. *Implementing Lean Software Development from Concept to Cash*. Reading (MA): Addison-Wesley, 2007.

Derby E, and Larsen D. ***AGILE** Retrospectives: Making Good Teams Great*. Pragmatic Bookshelf, 2006.

Highsmith J. **AGILE** *Project Management: Creating Innovative Products*. Reading (MA): Addison- Wesley, 2010.

Anderson DJ. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.

Duvall PM. *Continuous Integration: Improving Software Quality and Reducing Risk*. Reading (MA): Addison-Wesley, 2007.

3 Terms and definitions

For the purposes of this document and within its scope, the following terms and definitions apply. Rather than addressing or referencing all terms/definitions applicable to both the **AGILE** and regulatory perspectives, only those that are significantly pertinent to multiple TIR areas and/or need explanation between competing views have been included. Defined terms appear within the body of the document in bold **SMALL CAPS**.

3.1
AGILE

with respect to software or product development, term that does not denote a specific methodology, approach, or practice (i.e., there is no generally accepted, specific **AGILE** methodology) but rather an umbrella term that is typically applied when software or product development 1) fits with the spirit of the *Manifesto for **AGILE** Software Development*, see https://agilemanifesto.org/ [16], 2) is more empirical than deterministic, and 3) is **EVOLUTIONARY** and frequently iterative; the **AGILE** Alliance, see http://www.agilealliance.org, describes "**AGILE** Software Development" in the following manner: "*In the late 1990's several methodologies began to get increasing public attention. Each had a different combination of old ideas, new ideas, and transmuted old ideas. But they all emphasized close collaboration between the programmer team and business experts; face-to-face communication (as more efficient than written documentation); frequent delivery of new deployable business value; tight, self-organizing teams; and ways to craft the code and the team such that the inevitable requirements churn was not a crisis*." Definitions of the generic word **AGILE** usually include "quick," "well-coordinated," "adaptable," and other similar words and are qualities that typically pertain to **AGILE** development

3.2
ACCEPTANCE TEST-DRIVEN DEVELOPMENT; ATDD

form of **TEST-DRIVEN DEVELOPMENT** that concentrates on applying **TDD** at the feature or **STORY** level, based on acceptance tests developed for requirements at those levels; therefore, **ATDD** generally involves a cross-functional team test approach with users, customers, and others beyond quality assurance (QA) representatives and technical engineers/developers; **ATDD** may also involve **VALIDATION** in addition to **VERIFICATION**

3.3
BACKLOG

Set of work to be **DONE**; in **AGILE** terms, a **BACKLOG** is usually a list of user/customer-meaningful functionality or features that is ordered in terms of value and encompasses the breadth of the system or product to be built (i.e., is MECE, mutually exclusive and collective exhaustive), but not necessarily the depth in terms of detail; **BACKLOG**s should encompass all work that a team must do in order to facilitate effective prioritization and planning, and therefore can include functional and non-functional/infrastructure items as well as defects; also termed "product **BACKLOG**" when the **BACKLOG** describes all the product work to be **DONE**, and as "**ITERATION BACKLOG**" for work to be completed during a specific **ITERATION**

3.4

BUILD

operational version of a system or component that incorporates a specified subset of the capabilities that the final product will provide

3.5

BURNDOWN/BURNUP CHART

specific type of progress tracking chart that focuses attention on capabilities implemented/target value delivered (**BURNUP**) or capabilities left to implement/target value remaining (**BURNDOWN**); these types of charts can be used within any level (PRODUCT LAYER, **RELEASE**, **INCREMENT**, **STORY**) and can also be used for task hours tracking (work completed/remaining); because of the **AGILE** focus on results rather than activity, however, at least one level of progress chart should be on value-oriented progress; with iterative approaches, usually at least two levels of progress charts are maintained, one for the overall progress and one for particular **ITERATION**(s) underway; progress in value delivery is often termed "velocity" and is a key **AGILE**-related measure

3.6

CAPA

Corrective and Preventive Action

3.7

DESIGN INPUT/OUTPUT

as defined by the FDA quality system regulation, **DESIGN INPUTS** are "the physical and performance requirements of a device that are used as a basis for device design" (21 CFR 820(f)); it defines **DESIGN OUTPUTS** as "the results of a design effort at each design phase and at the end of the total design effort;" definition and management of **DESIGN INPUTS** and **DESIGN OUTPUTS** are central to medical device control and quality management systems; they also directly affect how **AGILE** approaches are tailored because controls are integral to choice and adaptation of lifecycle models

3.8

DONE

term usually referring to completion of all activities necessary to deliver usable software, with varying degrees of "**DONE**" applying to the **PRODUCT**, **RELEASE**, **INCREMENT**, and **STORY LAYERS**; the concept of "**DONE**" is critical in **AGILE** approaches; because of the emphasis on delivering usable software and achieving results over activity, defining what "**DONE**" means is central to defining an **AGILE** life cycle model, associated processes, and product/project success criteria; variations and phrases include "Definition of DONE," "**DONE DONE**," "**DONE** is **DONE** and "**DONE**ness" (as in degree of achieving "**DONE**")

3.9

EMERGENCE/EMERGENT

in **AGILE** contexts, a concept usually ascribed to a system that cannot (or is not desirable to) be defined fully in advance of doing development work, and where the actual conduct of work informs and refines both the definition of the system and the work process itself, providing both better system and work results

3.10

EVOLUTIONARY STRATEGY

life cycle model in which a system is developed in **BUILDS**; an **EVOLUTIONARY STRATEGY** is similar to an **INCREMENTAL** strategy but differs in acknowledging that the user need is not fully understood and all requirements cannot be defined up front; in this strategy, customer needs and system requirements are partially defined up front, then are refined in each succeeding **BUILD**.

Figure 1 graphical view of the **EVOLUTIONARY** approach, emphasizing the customer feedback that is an essential component of **AGILE**; [16]in this view, there is a preliminary requirements analysis (analogous to **BACKLOG** creation) and an initial high-level design activity, both of which are commonly found in the beginning stages of many **AGILE** projects with an emphasis on "just enough" to get started on the version **BUILD** cycle.

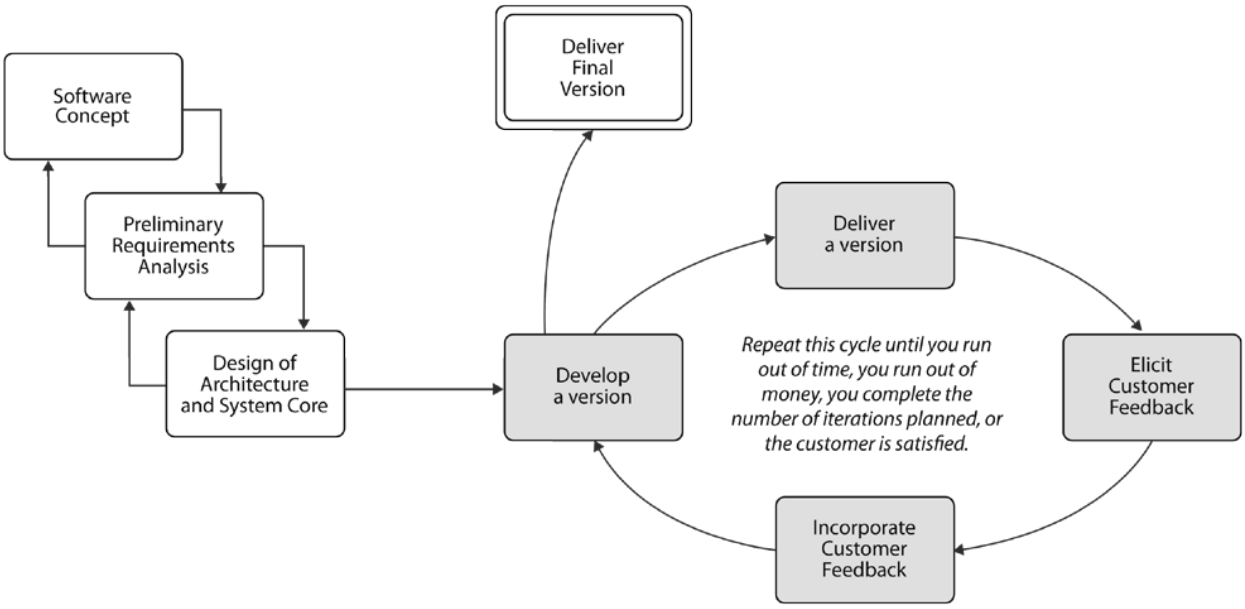

Figure 1—EVOLUTIONARY life cycle

3.11

EXECUTABLE REQUIREMENTS

requirements that enable automatic generation of associated software tests; currently, **EXECUTABLE REQUIREMENTS** are typically either 1) appropriately developed/formatted requirements that allow automatic generation of tests, or 2) using executable tests as both tests and requirements (**TEST-DRIVEN DEVELOPMENT** and its variants)

3.12

INCREMENT

portion of system functionality; the term also refers to the segments of development activity that result in **INCREMENTs**. An **INCREMENT** could be an increment of time, such as one of more **ITERATIONS**, or it could an **INCREMENT** of the product, such an incremental creation of a set of functionality. See **INCREMENTAL** and **ITERATION**.

3.13

INCREMENTAL

1) the "**INCREMENTAL**" strategy [life cycle model] determines customer needs and defines the system requirements, then performs the rest of the development in a sequence of **BUILDs** that INCREMENTALLY establish the intended system. The first **BUILD** incorporates part of the planned capabilities; the next **BUILD** adds more capabilities, and so on, until the system is complete

2) a software development technique in which requirements definition, design, implementation, and testing occur in an overlapping, iterative (rather than sequential) manner, resulting in **INCREMENTAL** completion of the overall software product

the key difference between the two definitions is the degree of requirements and design completeness before detailed design and **BUILD**; **AGILE** tends to be closer to the second definition and more similar in first steps (e.g., **BACKLOG** creation, **RELEASE** planning) to Figure 1, with a rejection of mandatory completion of requirements and high-level design definition upfront before other activity, as shown in Figure 2 and Figure 3; in most cases, the issue is what is being defined, when, and to what degree of detail; **AGILE** approaches often tend toward a combination of **INCREMENTAL**, **EVOLUTIONARY**, and iterative approaches in varying degrees

INCREMENTAL approaches also vary in terms of delivery, i.e., whether deliveries occur after one or more **BUILDs** or not until the entire system (or as much as practicable) is built; these variations are similar to life cycle model descriptions of "Evolutionary" and "Staged Delivery" (Figure 1 and Figure 2 where 1 to n deliveries are made) and "Design to

Schedule" (Figure 3 where a single delivery is made); although non-**AGILE** software groups commonly do not develop **INCREMENT**s in priority order, McConnell presaged **AGILE BACKLOG**s by referencing such prioritization

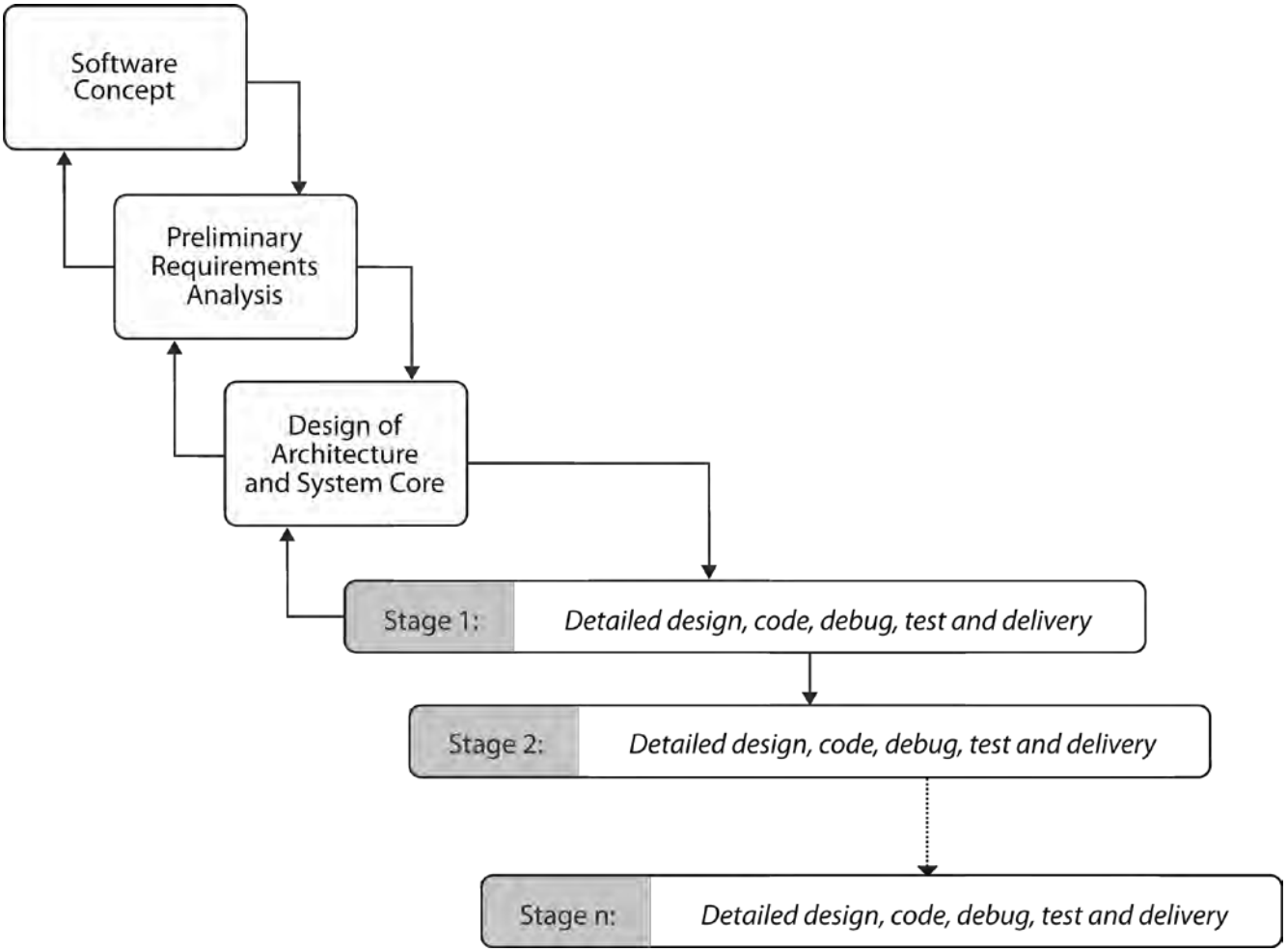

Figure 2—INCREMENTAL life cycle: "Staged delivery"

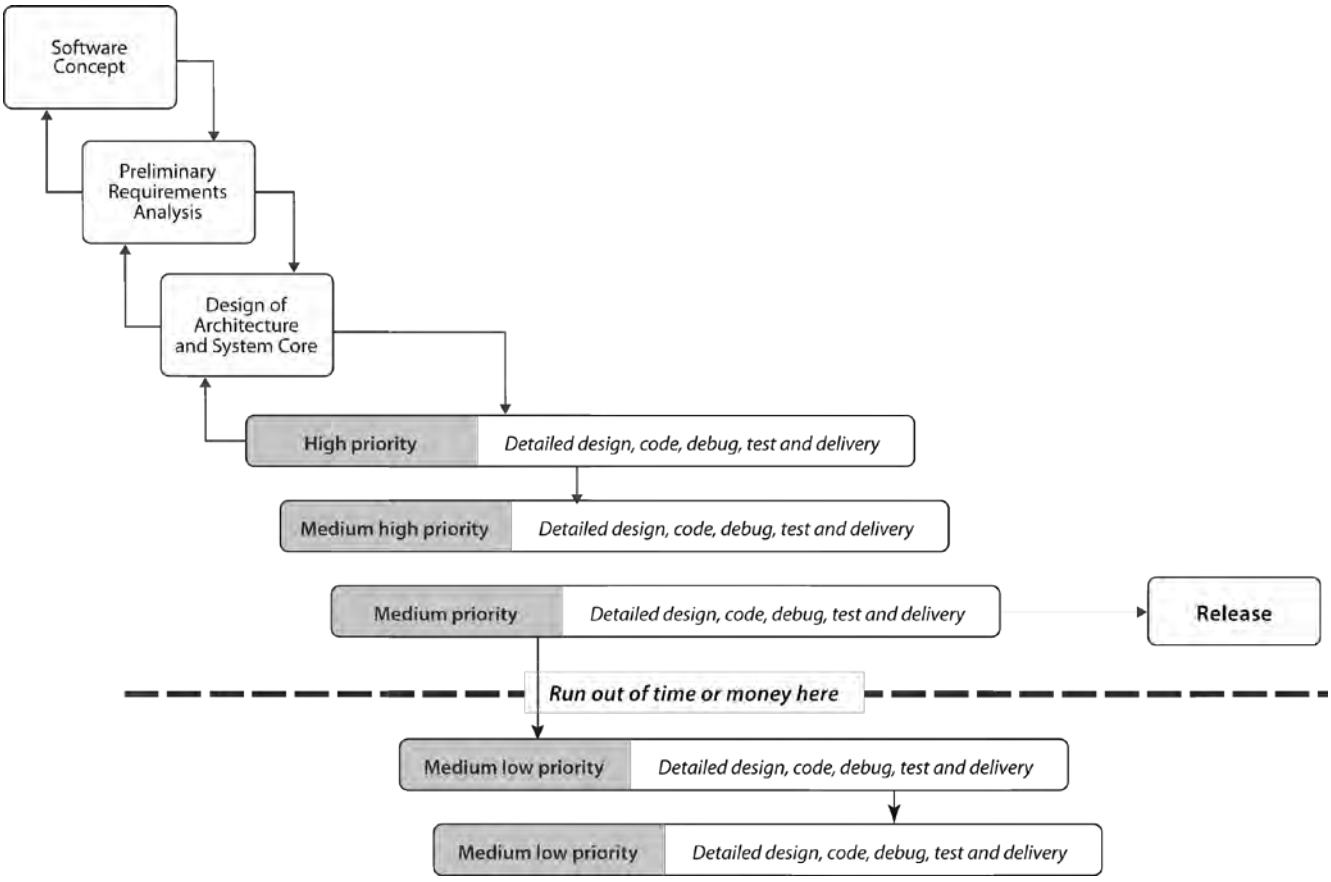

Figure 3—INCREMENTAL life cycle: "Design to schedule"

3.14

INCREMENT LAYER

see the description in Clause 5.1.3, "Executing process activities at multiple layers of abstraction"

3.15

ITERATION

in general terms, ITERATION can be defined as:

1) Process of performing a sequence of steps repeatedly;

2) Single execution of the sequence of steps in (1).

iterative development usually means repeating a life cycle model or a significant portion of the life cycle model multiple times in the course of a single project, rather than a single execution of a selected life cycle;

in **AGILE** frameworks such as Scrum, **ITERATION** means the repeated set of activities related to planning and delivering items from the **BACKLOG**; in Scrum's case, an **ITERATION** is termed a "Sprint" and is subject to Scrum's particular **ITERATION** rules/principles

3.16

LEAN

term popularly applied outside of Japan to the philosophy behind the Toyota Production System and other implementations; although definitions vary, just as with "**AGILE**," **LEAN** is commonly characterized by a focus on the following:

— Customer value;

— Waste reduction;

— Empirical thinking (application of scientific method);

— Sustainable flow and application of queuing theory to minimize work in process/idle inventory;

— Mastery (discovery, accumulation, and exploitation of knowledge overall and in terms of individual worker capabilities).

LEAN is not within the scope of this TIR, except for reference for reader knowledge because most **AGILE** approaches share, either explicitly or implicitly, elements of **LEAN** philosophy and the **LEAN** field provides additional material for the **AGILE** investigator

3.17
PRODUCT LAYER
see the description in Clause 5.1.3, “Executing process activities at multiple layers of abstraction.”

3.18
RELEASE
1) Delivered version of an application which may include all or part of an application.

2) Collection of new and/or changed configuration items which are tested and introduced into the live environment together.

3) Software version that is made formally available to a wider community.

Note to entry: The “wider community” might be internal to an organization or external to it.

4) Particular version of a configuration item that is made available for a specific purpose.

5) Formal notification and distribution of an approved version.

3.19
RELEASE LAYER
see the description in Clause 5.1.3, “Executing process activities at multiple layers of abstraction.”

3.20
REFACTOR/REFACTORING
improving the software, or reducing technical debt, without changing behavior or functionality; in other words, the end result/output of the software stays the same, but how the result is produced is changed or clarified

3.21
RETROSPECTIVE
meeting typically held at the end of an **ITERATION**, **INCREMENT**, or **RELEASE,** in which the team examines its processes to determine what succeeded and what could be improved; the **RETROSPECTIVE** is key to an **AGILE** team's ability to “inspect and adapt“ in the pursuit of “continuous improvement” the **AGILE RETROSPECTIVE** differs from other methodologies’ “lessons learned” exercises, in that the goal is not to generate a comprehensive list of what went wrong; a positive outcome for a **RETROSPECTIVE** is to identify one or two high-priority action items that the team wants to work on in the next **ITERATION**, **INCREMENT**, or **RELEASE**; the emphasis is on actionable items, not comprehensive analysis; **RETROSPECTIVES** take many forms, but there is usually a facilitator, who might or might not be a member of the team, and the process is typically broken down into three phases: data gathering, data analysis, and action items

3.22
SOFTWARE DEVELOPMENT LIFE CYCLE MODEL
conceptual structure spanning the life of the software from definition of its requirements to its **RELEASE** for manufacturing, which:

— identifies the process, activities and tasks involved in development of a software product;

— describes the sequence of and dependency between activities and tasks; and

— identifies the milestones at which the completeness of specified deliverables is verified. [2]

3.23

SOFTWARE OF UNKNOWN PROVENANCE

SOUP

software item that is already developed and generally available and that has not been developed for the purpose of being incorporated into the medical device (also known as “off-the-shelf software”) or software previously developed for which adequate records of the development processes are not available

3.24

STAKEHOLDER

a STAKEHOLDER loosely refers to anyone outside the Scrum team who has an interest in the product that the team is producing; STAKEHOLDERS can include but are not limited to direct managers, subject matter experts, account managers, salespeople, legal officers, and customers

3.25

STORY

“a user **STORY** is a short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. They typically follow a simple who/what/why template: *As a <type of user>, I want <some goal> so that <some reason>;*” the “why” is an important component in **AGILE** as it is a descriptor of the change’s value and helps retain focus on results important to the “who” rather than just how to achieve the goal. Also termed “user **STORY**”

STORIES typically include accompanying acceptance criteria; the objective of a **STORY** is to provide a persistent, lightweight requirement artifact in a form that 1) is understandable by all pertinent STAKEHOLDERS, 2) has just enough information to estimate the relative size of the **STORY**, 3) captures just enough essence as a placeholder for future discussions to uncover or elaborate more of the requirement when needed, but not before needed, and 4) enables change and elaboration with minimum overhead and waste

3.26

STORY LAYER

see the description in Clause 5.1.3, “Executing process activities in multiple layers of abstraction.”

3.27

TEST-DRIVEN DEVELOPMENT

TDD

generically, **TDD** reverses the **BUILD** and test process by first developing the test, checking that it fails (test works), and then iteratively developing small components of software (or product function) until the particular component under development passes the test(s); emphasis is on only developing functionality necessary to pass the specific test and nothing more. Once the initial test is passed, any improvement (**REFACTORING**) of the component is completed and then the tests reapplied. Once **REFACTORING** is complete, then the next small component or function will be developed

if a **TDD** approach is used, then the tests may also be used as requirements or significant portions of requirements, because the tests become the specifications for development; **TDD** began as primarily a software-oriented practice focusing on developer-driven **VERIFICATION**; now there are several variants of **TDD** like **ATDD** (see separate term entry) and behavior-driven development

3.28

TRACEABILITY

Degree to which a relationship can be established between two or more products of the development process

3.29
VALIDATION
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

3.30
VERIFICATION
confirmation through provision of objective evidence that specified requirements have been fulfilled

Note to entry: "Verified" is used to designate the corresponding status.

3.31
WATERFALL
"once-through" strategy [life cycle model] of performing the development process a single time. Simplistically: determine customer needs, define requirements, design the system, implement the system, test, fix and deliver

4 Setting the stage

This section provides a brief introduction about the **AGILE** and Regulatory perspectives. The rest of the TIR will compare and contrast these two perspectives, showing where they align well, where they may not, and provide suggestions to make them align. The references and bibliography provide good sources for more information to provide a deeper understanding of these two perspectives.

4.1 The AGILE perspective

AGILE software development is a set of principles and practices used by self-organizing cross-functional teams to rapidly and frequently deliver customer-valued software. It follows an **INCREMENTAL/EVOLUTIONARY** life cycle, emphasizing close collaboration between the software development team, the customer, and other STAKEHOLDERS. It is adaptable, emphasizing the need to adjust the principles and practices to fit the context and environment in which the software is being created.

4.1.1 AGILE benefits, values, principles, and practices

AGILE was created in response to the challenges posed by using a traditional, highly planned/managed, document-focused, **WATERFALL** method of software development. The early mechanisms of **AGILE** were referred to as "Lightweight Methodologies," a reaction to what was widely viewed as the unduly heavy practices that existed in the software development community.

The application of **AGILE** may provide the following benefits:

- Quality: Correctness of the product, meeting its specifications, free of defects, reliable;
- Productivity: Efficiency and speed of the development team, maximizing delivered value while minimizing development cost;
- Predictability: Improved estimation and planning, controlled and manageable response to change;
- Responsiveness: short cycle times that emphasize rapid learning, rapid adjustment to respond to emerging and changing needs of customers;
- Product effectiveness: meeting the customer's needs, usable.

Among medical device customers and STAKEHOLDERS, safety is arguably the highest valued characteristic of any product. Using an **AGILE** approach to the development of software does not inherently result in safer software, but it's use maximizes the delivery of software that meets **CUSTOMER/STAKEHOLDER** value.

The *Manifesto for **AGILE** Software Development* (https://agilemanifesto.org [16]) represents the high- level value statements that define what **AGILE** is and guide how it is adapted.

The Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work, we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more

Specific AGILE frameworks and methods (such as Scrum and the Scaled Agile Framework (SAFe®) are based on this manifesto and other LEAN- **AGILE** principles. Here are some **AGILE** principles that apply to the context of medical device software that will be addressed by this TIR:

- Apply **INCREMENTAL** and **EVOLUTIONARY** life cycle concepts, with **INCREMENT**s being as short as a week and evolution happening as late as the last **INCREMENT** before a product is shipped.
- Define what "**DONE**" means and use that definition to get to "**DONE**" at each boundary of the **INCREMENTAL** life cycle.
- Deliver customer value, with the highest priority features first, through close collaboration between the customer and the project team.
- Accept that customer needs and requirements are likely to change, so accommodate this in an efficient and effective way.
- Manage project risk through increased visibility of progress and stronger team accountability.
- Use self-organized teams empowered to manage the daily tasks of producing software.
- Seek technical excellence in the software through high-quality designs and thorough **VERIFICATION** practices.
- Reflect at regular intervals and adapt the process to constantly improve quality and efficiency of work.
- Satisfy business STAKEHOLDERS, both internal and external.

There are many different ways to implement **AGILE** (e.g., Extreme Programming [XP], Scrum, Feature-Driven Development [FDD], and others), each one defined by its collection of **AGILE** practices. Rather than focus on a specific type, this TIR aggregates various practices into a set that is representative of a broader view of **AGILE** practices. The following list is not a complete collection of all **AGILE** practices but rather a list of practices that apply to the context of medical device software that will be addressed by this TIR.

- Practices related to defining the product, including product vision/vision statement, product **BACKLOG**, **STORIES**, use cases/personas, and **BACKLOG** management.
- Practices related to design, construction, and test, including **EMERGENT**/**EVOLUTIONARY** design/**REFACTORING**/architecture spikes, coding guidelines/standards, **TDD**, continuous integration, daily **BUILD**s/automated **BUILD**s, frequent delivery/frequent **RELEASE**s, unit testing, **STORY** testing/**EXECUTABLE REQUIREMENTS**, and test automation.

— Practices related to project execution, including time-boxing/fixed **INCREMENT** lengths, **RELEASE** planning, **INCREMENT** planning, daily planning, "Definition of DONE", and velocity/**BURNDOWN/BURNUP CHART**s.

— Practices related to team organization, including self-organizing team, team roles, team size, scrum of scrums, and sustainable pace.

— Practices related to team collaboration, including stop-the-line/information radiators, co-location, pairing, reflections/**RETROSPECTIVE**s, and collective ownership.

For more information, see Annex A.

4.2 The regulatory perspective

Regulation of medical devices is intended to protect the public health, by assuring that marketed products are safe, effective and secure. Different countries and regions have different agencies that regulate medical devices. While these different agencies have different regulations, they share much in common and all have similar goals and guiding principles. This section concentrates on regulation for quality system requirements specific to design control.

4.2.1 The United States FDA regulatory perspective

In the United States, the Food and Drug Administration (FDA) is responsible for regulating medical device software, using a set of laws, regulations, and guidance documents.

The Federal Food, Drug, and Cosmetic Act (FD&C Act) is a federal law enacted by Congress. It and other federal laws establish the legal framework within which the FDA operates. The FFD&C Act can be found in the United States Code, which contains all general and permanent U.S. laws, beginning at 21 U.S.C. 301.

In order to enact the requirements, set forth in the law, the FDA develops regulations based on the laws set forth in the FD&C Act or other laws under which the FDA operates. FDA regulations are also federal laws, though they are not part of the FD&C Act. The 1990 Safe Medical Device Act established Pre-Production Design Controls, which led to the Quality System Regulation 21 CFR 820 (the QSR).

Subpart C (820.30) of the QSR, Design Controls, requires manufacturers to establish and maintain procedures to control the design of the device to ensure that specified design requirements are met. Section C allows the manufacturer to choose the development process they will use, the development methodology and the specific software development life cycle. While it does not require a specific development methodology, it does require that all elements of Design Controls are addressed.

Figure 4 shows some of the elements of the Design Controls from 21 CFR 820.30.

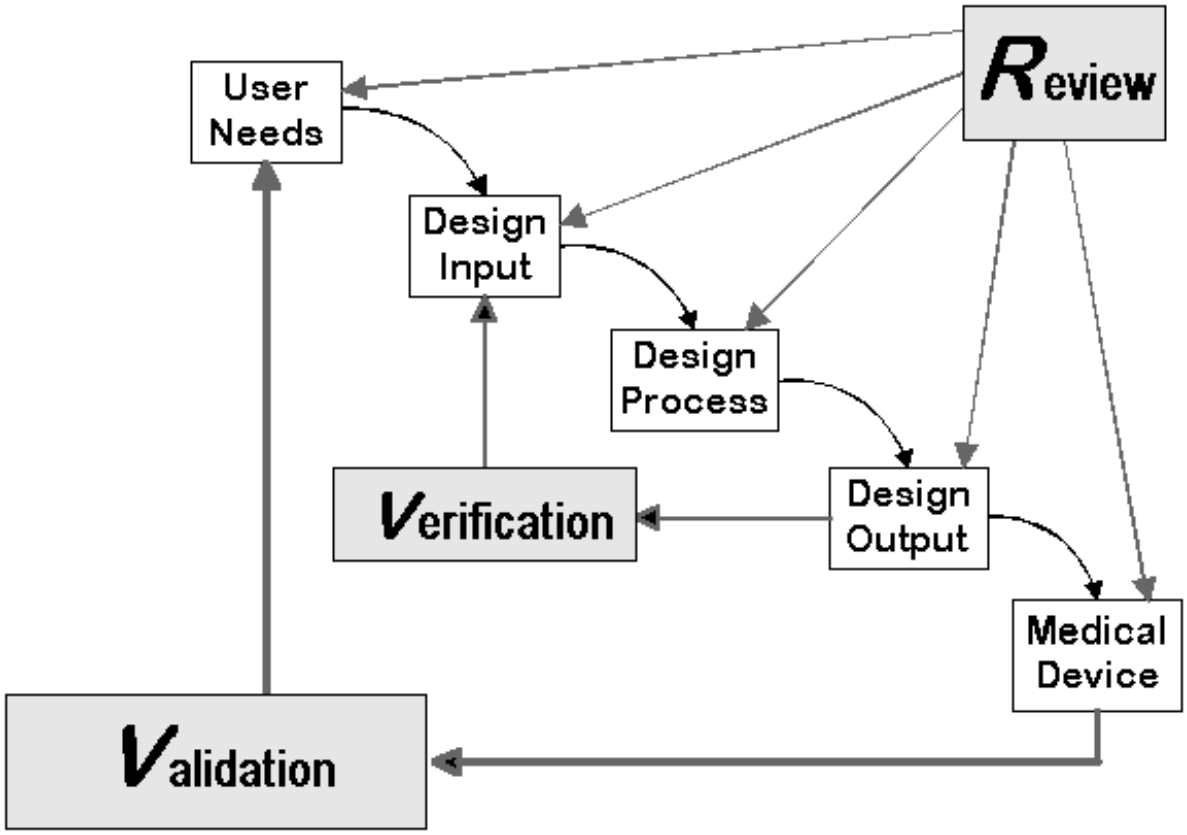

Figure 4—Design Controls from 21 CFR 820.30

Note to entry: From the "FDA's Design Control Guidance For Medical Device Manufacturers" - https://www.fda.gov/media/116573/download

While this diagram gives the impression of a linear or **WATERFALL** life cycle, the guidance does not prescribe a particular life cycle model, allowing the manufacturer to choose which life cycle model they will use.

To describe the agency's current thinking on the law or the regulations, the FDA frequently publishes guidance documents. Guidance is not legally binding on the public or the FDA, but it sets expectations and provides insight and knowledge about how certain regulations need to be applied to medical devices and medical device software.

4.2.2 European Union (EU) and other regulatory perspectives

Internationally, regulatory agencies in key countries and regions have recognized the requirements contained in ISO 13485 (*Medical devices—Quality management systems—Requirements for regulatory purposes*) and have incorporated these requirements into their quality system regulations. ISO 13485 is a recognized standard specifying requirements for a comprehensive management system for the design and manufacture of medical devices.

Subclauses of ISO 13485 Clause 7.3 (Design and Development) have the same overall objectives of the United States FDA regulation related to design controls. For some time the FDA, Health Canada, the European Union (EU), Japan, and Australia have been working within the International Medical Device Regulators Forum (IMDRF) to develop guidance documents that reflect international agreement on quality management system essential principles and requirements. Information on guidance documents from the various GHTF study groups can be found at http://www.imdrf.org/.

4.2.3 IEC 62304 standard

IEC 62304 is an international standard that defines the development and maintenance life cycle requirements for medical device software. The set of processes, activities, and tasks described in 62304 establishes a common

framework for medical device software life cycle processes. While 62304 is not itself a regulation, it is recognized by many regulatory agencies as the DeFacto standard for developing medical device software.

Similar to the FDA's Design Controls (820.30), the IEC 62304 standard presents information in what appears to be a linear model, but 62304 explicitly states it does not prescribe a specific life cycle model, only that one must be defined and shown to address the required elements of the standard.

4.2.4 Regulatory goals, values, principles, and practices

Regulatory agencies are responsible for protecting the public health by assuring the safety, efficacy, and security of medical devices.

The application of regulations may provide the following benefits:

- Quality: correctness of the product, meeting its specifications, free of defects, reliable;
- Safety: identifying and mitigating safety risks to patients and users;
- Effectiveness: meeting the customer's needs, usable, delivering value to the patient and users.

While regulations and standards do not seek benefits in productivity or project predictability, they don't preclude them from being important to medical device software manufacturers.

Unlike the **AGILE** perspective, the regulatory perspective does not have a specific manifesto that describes its driving value statements, but, similar to the **AGILE** perspective, values and principles can be derived from the regulations, standards, and guidance documents. The following are some principles that are relevant to the use of **AGILE** for medical device software that will be addressed by this TIR. The intended use and the safety associated with the software to be developed should be considered when determining the degree of control and the level of effort to be applied.

- Established (controlled) processes are essential for producing high-quality software; in other words, it is important to establish processes that are intended to provide control over design and development processes. The degree of control and the level of effort to be applied should be based on the intended use and the safety associated with the software to be developed.
- Regulatory agencies do not prohibit or encourage the use of any specific software development methodology, but they do indicate some expected characteristics of the selected software life cycle and development.
- A Quality Management system is a formalized system of processes, procedures and responsibilities for meeting quality processes and objectives. Medical Device quality systems allow organizations composed of staff with appropriate training and direction to meet customer and regulatory requirements.
- Documentation (evidence) is necessary to demonstrate compliance to regulations, and to facilitate the investigation into software problems, as well as to evaluate software for those devices requiring regulatory approval, e.g., approval, clearance, licensing, registration, self-certification, etc.
- Planning is important to ensure effective execution of processes and high quality in the software product.

Regulatory requirements, guidance documents, and standards provide a framework for software development practices, identifying activities that must be performed.

Broad categories of development practices that are used to meet regulation include:

- Patient safety and risk management practices, including the identification of risk scenarios, mitigations for those risks, and **VERIFICATION** activities to assess the remaining risk in a software product;

- Quality planning practices, including management controls, software development environment planning, and team training;
- Documentation practices, including document controls, configuration management, and change controls;
- Software definition practices, including user-level requirements, design-level requirements, and system and software interface definition;
- Software design practices, including architectural design, detailed design, implementation, and design change management;
- **VERIFICATION** and **VALIDATION** practices, including design reviews, peer reviews, unit-level testing, integration testing, system testing, and user **VALIDATION** testing.

Regulatory documents often do not get into the details of how these practices must be performed, leaving those details up to the manufacturer to define. This approach allows manufacturers to create their own practices that align with regulatory principles.

4.2.5 Where to learn more

Refer to Clause 2 “Normative references”, for a list of regulations, standards, and guidance documents.

4.3 Aligning perspectives

Before getting into the detailed topics in Clause 5 and beyond, this section will explore where the **AGILE** perspective and the regulatory perspective align.

4.3.1 Aligning on goals

The **AGILE** and regulatory perspectives align well on their goals for quality and customer satisfaction.

Both the **AGILE** world and the regulated medical device software world clearly place high value on the quality of the software that is produced. While publications from either perspective might use different words to reflect their particular emphasis, the goals are well aligned. For example, the regulatory perspective of quality focuses on safety and performance, while the **AGILE** perspective focuses on the more general goal of meeting the needs of the **CUSTOMER**. These are simply different ways to emphasize compatible goals.

AGILE *and regulatory perspectives both value high-quality software.*

The **AGILE** perspective puts greater emphasis on the goals of increasing product development productivity and predictability than the regulatory perspective does, but this does not make them incompatible. Improving productivity and predictability are worthy goals for any development organization.

Align the goals of quality, productivity, and predictability to be supportive of one another.

An over-emphasis of one goal might suggest the de-emphasis of another. This is rarely necessary or desirable. It is better to pursue improvements that benefit all the goals together. Rather than say, “We are going to improve productivity by eliminating a process step without regard to what it does to quality,” we should say, “We are going to reduce the effort of some activities that don’t enhance quality so we can spend more effort on activities that do.”

4.3.2 Aligning on values

The values of the **AGILE** and regulatory perspectives actually align rather well, though it may be hard to recognize that at first. The **AGILE** Manifesto is a clear, powerful, and intentionally provocative statement of the values that drive **AGILE** software development. While there is no specific manifesto that describes regulatory values, the values that can be derived from regulations, standards and guidance documents are also powerful drivers of the Regulatory perspective on software development.

The last phrase of the manifesto provides a vital qualification of the **AGILE** values: "while there is value in the items on the right, we value the items on the left more." Thoughtful consideration to this phrase is essential to the proper application of **AGILE** in the medical device software world, or, in fact, any environment.

Whether it is due to being regulated, or due to the industry's own strong beliefs, the medical device software world clearly finds value in processes and tools, documentation, contracts, and plans. The **AGILE** Manifesto places more value on: Individuals and interactions, working software, customer collaboration and responding to change, and these are also valuable to the medical device software world. The value statements in the **AGILE** Manifesto need not, and should not, be viewed as being two mutually exclusive choices. Instead, when applied properly using the recommendations provided in this TIR, they should be used as guides to find the proper balance.

Find the proper balance of what is valued.

Beyond simply recognizing that the values of the **AGILE** Manifesto are not incompatible with regulatory values, each of the statements provides useful concepts that bring benefit to the development of medical device software.

For a more detailed examination of these topics, see Appendix A.

4.3.3 Aligning AGILE with a quality management system

A foundational element of regulations and standards for medical device software is that a quality management system must be "established," where "established" means that the quality management system is defined, documented, and implemented. It must be understood by those who use it, and objective evidence must be produced to demonstrate it is being properly used. A quality management system is documented within the organization's common process documentation (such as procedures, protocols, and work instructions) as well as specific documentation unique to a project (such as in project plans and reports).

AGILE provides a set of principles and practices for developing software with emphasis on techniques for planning and executing a development effort, but it does not provide a complete definition of the process for developing software. To produce safe and effective medical device software, and to satisfy regulatory requirements, **AGILE** should be applied within the context of an established quality management system.

Apply **AGILE** *within the context of an established quality management system.*

The use of **AGILE** may have little impact on the Quality Management System from a regulatory perspective. Regulatory submissions might look the same, other than details written in the project plan, so at the high level of a normal submission, the details of an **AGILE** implementation might not be apparent or relevant. Even in a compliance audit, depending on the scrutiny given to software development, auditors might not get into details relevant to the implementation of **AGILE**.

Documentation might also look the same. If the quality management system requires documentation that is familiar and useful to development teams and regulators, the introduction of **AGILE** need not change that documentation. That documentation may be produced differently using **AGILE** principles and practices (see section 5.5 Documentation), but the finished product may not look any different than it did before **AGILE** was introduced.

Changes to the Quality Management System might be required if the current system is incompatible with **AGILE** concepts. For example, if the quality management system mandates or assumes a more linear, **WATERFALL** life cycle, some changes will be necessary to accommodate **AGILE'S INCREMENTAL-EVOLUTIONARY** life cycle.

When **AGILE** *is used in the context of an established quality management system,* **AGILE** *need not disrupt the established system. Changes that* **AGILE** *practices might require should be shown as enhancements to the quality management system.*

Changes that **AGILE** brings to a robust and effective quality management system should not diminish or give the perception of diminishing the effectiveness of the established system. On the contrary, changes should be made within

the requirements of regulations, regulatory guidance, and standards, while demonstrating the value that **AGILE** methods bring in enhancing the effectiveness of the quality management system.

4.3.4 Aligning on the level of rigor and robustness

Regulatory requirements and guidance documents recognize that different kinds of medical device software require different development processes, practices, and documentation. For example, the FDA's "Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices" describes "Level of Concern," recommending that the extent of documentation to be submitted should be proportional to the Level of Concern associated with the device. The FDA's guidance document "General Principles of Software Validation" recommends that the specific approach, combination of techniques to be used, and level of effort applied to software development be based on the intended use and the safety risk associated with the software. IEC 62304 describes "Software Safety Classification" and provides guidance on the development processes to be applied depending on the classification. All of these are recommending that the risk associated with a software product should be assessed in order to establish a development process with the appropriate level of rigor and robustness.

The quality system processes and the mechanisms of **AGILE** *should provide the appropriate level of rigor and robustness to meet regulatory expectations.*

While different **AGILE** frameworks provide specific practices and mechanisms, they are designed to be flexible in their implementation, requiring an organization to regularly inspect and adapt the process and adapt it to their context.

5 Aligning on concepts

This section addresses some of the broad concepts to be considered when applying **AGILE** to medical device software.

AGILE provides many detailed practices but emphasizes the need to adapt them to fit the needs of a particular context, so it is important to understand the principles and foundational concepts of **AGILE** when adapting the practices. In contrast, regulations and standards provide broad guidance on the requirements for a quality management system but require manufacturers to provide the details that describe their process, so it is important to understand regulatory principles and foundational concepts when defining a quality management system. To align the **AGILE** and regulatory perspectives, the principles and foundational concepts need to be aligned. This section provides recommendations on how to achieve that alignment in a supportive way.

5.1 INCREMENTAL/EVOLUTIONARY life cycle

IEC 62304, Section B.1.1, describes three common life cycle models: **WATERFALL, INCREMENTAL**, and **EVOLUTIONARY**. These models have some elements in common and, in practice, the distinctions can be blurred when organizations implement a life cycle that is a hybrid of these common models. The primary difference between these models is the sequence in which activities are performed.

AGILE has elements of both **INCREMENTAL** and **EVOLUTIONARY** life cycles, rejecting the concepts of the **WATERFALL** life cycle. It follows an **INCREMENTAL** model in the way the broad product definition is broken into small pieces on the **BACKLOG**, with those pieces being further defined, designed, coded, and tested, a **STORY**-at-a-time, in short **INCREMENTS**. It follows an **EVOLUTIONARY** model in the ways it allows product definition to evolve over time, sometimes performing definition, design, coding, and testing activities in a highly overlapping and dynamic way. It rejects the **WATERFALL** model in that it discourages detailed long-term planning and the rigid staged-phases approach, preferring instead to have broad plans and product definitions for the long term, letting the detailed plans and activities evolve in a dynamic way as the near- term work is performed.

AGILE'S INCREMENTAL/EVOLUTIONARY life cycle executes process activities and produces outputs using a "sum-of-the-parts" model. Process activities are performed on small pieces of the software system, advancing the state of process outputs a small piece at a time. Pieces are integrated to demonstrate that the sum of the parts adds up to a consistent and correct whole.

AGILE *follows an* **INCREMENTAL/EVOLUTIONARY** *life cycle model, which might have a significant impact on a quality management system.*

The choice of an **INCREMENTAL/EVOLUTIONARY** life cycle has several significant impacts on a quality management system, some of which are addressed in the following sections.

5.1.1 Specifying the software development life cycle

Regulations (such as 21 CFR 820.30, Design Controls), guidance documents (such as General *principles of software validation,* Section 4.4, and Design Control Guidance for Medical Device Manufacturers Section III), and software development standards (such as IEC 62304, Section 5.1.1) require that medical device software manufacturers choose and define their **SOFTWARE DEVELOPMENT LIFE CYCLE MODEL**. They neither mandate nor prefer any particular model; instead, they require the manufacturer to choose the model they believe to be most appropriate to their situation.

Guidance documents and standards describe activities that a software process must address, such as product definition and requirements specification, high-level design, software coding, and testing. They provide requirements for performing those activities, some of which address the relationships between activities, such as input/output relationships (e.g., requirements are an input to testing) and completion criteria requirements (e.g., configuration items must be controlled before **VERIFICATION** can be completed). Because these process descriptions are often organized in a sequence that looks like a traditional **WATERFALL** model, it can be mistakenly assumed that a **WATERFALL** life cycle is the expected model. Because **AGILE** is a highly **INCREMENTAL/EVOLUTIONARY** approach, it can therefore be mistakenly assumed that **AGILE** is incompatible with the expectations for a medical device software process. For organizations that are more comfortable with linear life cycle models or have interpreted regulations and standards as implying a linear life cycle, an **INCREMENTAL/EVOLUTIONARY** life cycle can present challenges. Furthermore, for regulators or auditors who are more comfortable with linear life cycle models, an **INCREMENTAL/EVOLUTIONARY** life cycle might not meet their expectations.

To avoid misunderstandings, incorrect assumptions, or differing expectations, it is important for software development plans and procedures to define the life cycle to be used. Some sources of misunderstanding that should be addressed include the following:

- The process activities to be executed.
- The timing and sequencing of those activities.
- The inputs used by and the outputs generated by an activity.
- The process artifacts (the audit trail) to be produced.
- How the life cycle integrates with risk management.
- How change management and configuration management are implemented.

Define the life cycle model to set the proper expectations among the software development team and regulatory STAKEHOLDERS.

5.1.2 Mapping the process model to the life cycle model

Software development consists of a well-established set of activities that define the process model. IEC 62304 defines a process model for the activities necessary to develop medical device software. These activities can be executed in many different life cycle models, as IEC 62304, Section B.1.1, describes. When choosing to follow the activities defined by IEC 62304 in an **INCREMENTAL/EVOLUTIONARY** life cycle such as **AGILE**, it is important to define how the activities will be performed within that life cycle. This section describes how the activities defined by IEC 62304 are executed in **AGILE**'S **INCREMENTAL/EVOLUTIONARY** life cycle.

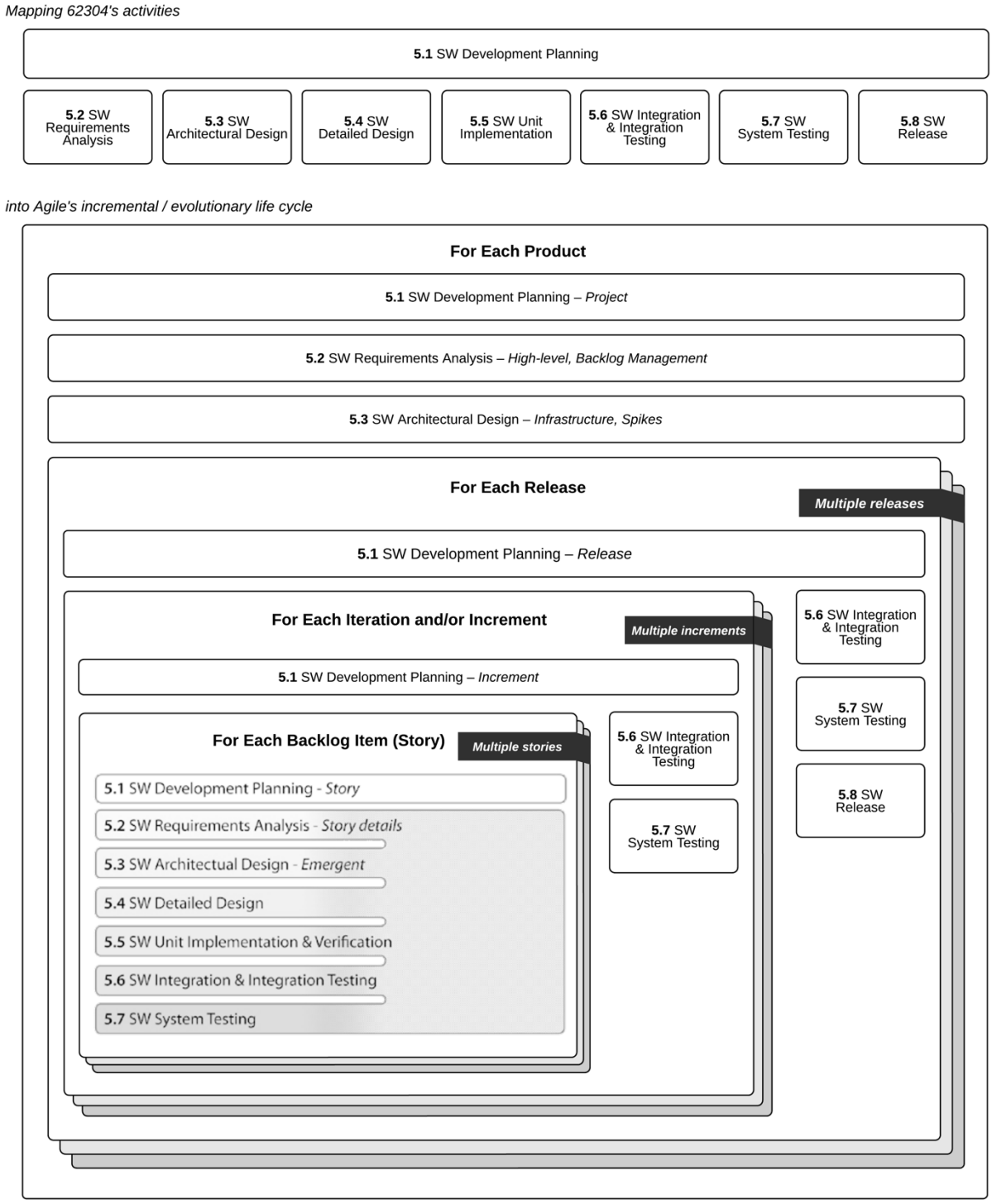

Figure 5—Mapping IEC 62304's activities into AGILE'S INCREMENTAL/EVOLUTIONARY life cycle

At the top of Figure 5 are the activities defined in IEC 62304, Section 5, Software development process. Although these activities are shown in a linear sequence that is representative of a **WATERFALL** life cycle, IEC 62304 does not specify any particular life cycle, and this figure is not meant to imply one.

The rest of Figure 5 shows a way for these activities to be executed in an **INCREMENTAL/EVOLUTIONARY** life cycle such as **AGILE**. The major groupings of activities are defined by the large, black-outlined boxes.

When defining the life cycle model, describe clearly how process activities are executed in that life cycle.

5.1.3 Executing process activities in multiple layers of abstraction

In **AGILE**'S **INCREMENTAL**/**EVOLUTIONARY** life cycle model, there are four layers of abstraction in which activities are performed, as shown in Figure 5.

The **PRODUCT LAYER** consists of the complete set of activities related to the development and delivery of a software product. (This might also be thought of as the "Project Layer" for organizations that organize they their AGILE activities by Project.) The **RELEASE LAYER** consists of activities to create a usable product. A product is delivered in one or more **RELEASE**S. Depending on the plan for the product, a **RELEASE** might generate a version of the product that is or could be shipped to the field, or it might generate a completed, consistent set of functionalities intended only for internal use. A **RELEASE** could span one to several months.

The **INCREMENT LAYER** consists of activities to create a set of useful functionalities, although not necessarily a complete software product. A **RELEASE** is made up of one or more **INCREMENT**S. An **INCREMENT** generally spans one to several weeks.

The **STORY LAYER** consists of activities to create a small piece of functionality, although not necessarily a complete set. An **INCREMENT** is made up of one or more **STORIES**. A **STORY** generally spans one to several days.

Figure 5 shows the process activities that are executed in each layer. Notice how most process activities are executed in more than one layer.

Software development planning happens in multiple layers. In the **PRODUCT LAYER,** planning addresses the high-level activities of project management: scoping the effort, forming teams, and gathering resources, planning the long-range duration and cost, and organizing the product into **RELEASE**S. In the **RELEASE LAYER**, planning addresses the mid-level activities of project management: scoping the **RELEASE**, establishing high- level integration points of the software and other subsystems, scheduling at the middle level, and organizing the **RELEASE** into **INCREMENT**S. In the **INCREMENT LAYER**, planning addresses the low-level activities of project management: planning and scheduling at the low level, establishing low-level integration points within software subsystems or other subsystems, and organizing an **INCREMENT** into **STORIES**. In the **STORY LAYER**, planning addresses detailed team and individual planning: assigning tasks, planning execution details for daily tasks, and tracking progress. Feedback occurs within and between all layers of planning.

Software requirements analysis happens in two layers of abstraction. In the **PRODUCT LAYER,** using the product **BACKLOG** mechanism, **DESIGN INPUTS** for the system are translated into high-level definitions of the software functionality. In the **STORY LAYER**, detailed specifications for the software are created. TRACEABILITY between the **DESIGN INPUTS** and software specifications are created. Software requirements analysis does not happen in the **INCREMENT LAYER** or **RELEASE LAYER**, although the planning activities of these layers will address which requirements are within the scope of the **RELEASE**S, **INCREMENT**S, and **STORIES**.

Software architectural design similarly happens in two layers of abstraction. In the **PRODUCT LAYER,** major architectural design happens. This might mean an entire **INCREMENT** is devoted to establishing the software's architectural infrastructure, or it might mean that smaller pieces of the architecture are developed via a set of developer-facing tasks not related to a specific **STORY** or **INCREMENT**. In the **STORY LAYER**, using the **AGILE** concepts of simple design and **REFACTORING**, the software architecture emerges as it needs to in order to support the new functionality being added by a **STORY**. Planning activities will address when architecture-related activities are planned for an **INCREMENT** and **RELEASE** and will determine whether those activities are to be performed as part of **STORIES** or as part of separate, non-**STORY**-related tasks.

Software detailed design and software unit implementation and **VERIFICATION** happens in the **STORY LAYER**. Using the concepts of simple design and **REFACTORING**, detailed design emerges as the code is developed. Using the concepts of **TDD**, unit-level tests are designed and executed as the code is developed.

Software integration and integration testing happens in three layers of abstraction. In the **STORY LAYER**, using the concepts of continuous integration, new functionality is integrated into the broader software product as the code is developed. Using the concepts of **TDD**, that integration is tested as the pieces are put together. In the **INCREMENT LAYER**, as individual **STORIES** are completed to create a complete set of related functionalities, using the concepts of **TDD**, that integration is tested as the code is developed. Similarly, in the **RELEASE LAYER**, as complete sets of functionalities are integrated with other complete sets or as the software is integrated with other subsystems, that integration is tested.

Software system testing also happens in those same three layers of abstraction. In the **STORY LAYER**, as a requirement is defined, a test (or other **VERIFICATION** method) is defined to verify it. Depending on what is needed to execute that test, the test might be executed in the **STORY LAYER** or it might not be executed until required elements of the system are integrated in the **INCREMENT LAYER** or **RELEASE LAYER**. In the **INCREMENT LAYER**, tests defined by a **STORY** might be executed or new tests might be created to verify requirements related to subsystem integrations. Similarly, in the **RELEASE LAYER**, existing tests might be re-executed, or new tests might be created to verify system-level integration requirements.

Software **RELEASE** activities happen entirely in the **RELEASE LAYER**.

*AGILE executes software development activities in different layers of abstraction, the **STORY LAYER**, the **INCREMENT LAYER**, the **RELEASE LAYER**, and the **PRODUCT LAYER**.*

5.1.4 Process flow: the timing and sequence of process activity execution

The software development activities of IEC 62304 are shown in Figure 5 as a linear sequence, although IEC 62304 does not imply that a linear life cycle must be chosen. In **AGILE'S INCREMENTAL/EVOLUTIONARY** process, activities are "turned 90 degrees," showing that the activities are executed in parallel with each other. Although not demonstrated by the diagram, in actuality, the process activities can also be executed in any imaginable combination of sequence, parallelism, and feedback loops. The shading of the activities in the **STORY** section shows that the activities can sometimes blur together into an indistinguishable group.

TEST – DRIVEN DEVELOPMENT (**TDD**) concepts suggest that testing activities should sometimes come before definition, design, and implementation. Rather than say, "Here are the requirements, now figure out how to test it," we might say, "Let's decide how we are going to test it, which will help us articulate the requirements." Similarly, rather than say, "Here is the unit, now write the tests to show it works," we might say, "Here is the test that will demonstrate how this unit must work, now build the unit to pass that test." Creating tests early in the development cycle means that the tests themselves become another form for specifying the desired behavior of the software. **TDD** emphasizes the creation of a robust set of continuously running tests. In addition to demonstrating code quality, these tests will highlight inconsistencies or unintended consequences of changes to help manage the dynamic nature of **INCREMENTAL/EVOLUTIONARY** development.

EMERGENT design and definition concepts suggest that design and implementation activities should sometimes come before definition. This is particularly useful when the customers are not sure of their requirements or when there are many solution choices. Rather than say, "Here are the requirements, now go make it," we might say, "Make me something that provides a few options, and then we can decide which paths to pursue further."

To avoid chaos and ensure that process activities are executed properly, planning is essential. Within each layer of the **STORY**, **INCREMENT**, and **RELEASE**, planning should specify the process activities to be completed. If there are specific timing or sequencing considerations for some activities, the plans should specify that as well.

Software development plans should specify which activities are performed within the ***STORY, INCREMENT, RELEASE,*** *and* ***PRODUCT LAYERS****.*

For more information about process flow, sequencing, and planning, see Clause 6.1.1.

5.1.5 Frequency and granularity of process activities

In a strict linear life cycle, a process activity could be executed once or perhaps only a few times. In an **INCREMENTAL/EVOLUTIONARY** life cycle, activities will be executed multiple times at multiple levels of granularity. Depending on the size of the development effort and the degree of granularity of the **STORIES**, any given process activity could be executed hundreds or thousands of times.

In the **STORY LAYER**, process activities are performed on a small part of the overall software system, the smallest part that is necessary to deliver the functionality defined to be in the scope of the **STORY**.

Requirements analysis activities might focus on a few software requirements, a single software requirement, or even just a portion of a software requirement, which might satisfy a few higher-level system requirements or only a portion of a higher-level system requirement or customer requirement.

Software architectural design and software detailed design focuses only on the architecture and design that is affected by the scope of the **STORY**.

Software unit implementation and **VERIFICATION** activities focus only on the narrowest elements of the code needed for the scope of the **STORY**, to satisfy the requirements.

Software integration and integration testing focus only on the integration of the new code added through the **STORY** with the software **BUILD**.

Software system testing focuses only on demonstrating that the affected software requirements have been verified.

Similarly, in the **INCREMENT LAYER**, **RELEASE LAYER**, and **PRODUCT LAYER**, process activities are performed as needed for the scope of the **INCREMENT**, **RELEASE**, or **PRODUCT**. The development planning activities within each layer should specify the scope of the process activities to be completed before the **INCREMENT**, **RELEASE**, or **PRODUCT** can be called complete.

*Process activities are executed many times in an **AGILE** model, at varying levels of granularity.*

5.1.6 The importance of integration activities

When process activities are performed small bits at a time, there is risk that all criteria for a process activity will not be completely satisfied one bit at a time. To mitigate this risk, which is inherent to an **INCREMENTAL/EVOLUTIONARY** life cycle, process activities are often executed at two different levels of granularity. At the low level, process activities focus on the small bit, ensuring that the process criteria for that bit have been satisfied. This is the focus of much of the work performed in the **STORY LAYER**. At the high level, process activities focus on the integration of the bits, ensuring that the process criteria for the complete set have been satisfied. This is the focus of some **STORY** work (**STORIES** whose purpose is to integrate other **STORY** work) and the integration activities in the **INCREMENT** and **RELEASE LAYERS**. For example, a requirements' process typically has some criteria (such as testability, clarity, and **TRACEABILITY**) that are best applied to each requirement as it is created, whereas other criteria (such as consistency, avoidance of contradiction, and document structure) are best applied to a broad set of requirements.

*Integration activities within the different layers (**STORY**, **INCREMENT**, and **RELEASE**) are important in demonstrating that highly **INCREMENTAL** development results in a complete and consistent product.*

For more information about integration activities, see Clause 6.1.6.

5.1.7 The importance of software configuration management

Traditional software development methodologies, such as phase-gated **WATERFALL,** were optimized for simplicity and visibility, requiring only a moderate level of discipline to manage the software system configuration throughout the projects. Keeping the development process simple allowed project teams to ensure that the right work products were baselined and controlled. A common traditional practice is to "freeze" the software content at a certain point in the project, not allowing any changes from that point forward.

AGILE approaches to software development require significantly more discipline in managing the software system configuration because work products are changing frequently. The value of responding to customer requirements changes is accompanied by the challenge of maintaining the software system configuration or baseline.

The software configuration system must be robust in handling frequent change.

For more information about configuration, see Clause 6.1.9.

5.1.8 Defining "DONE"

AGILE'S **INCREMENTAL/EVOLUTIONARY** life cycle delivers new functionality small bits at a time through **STORIES**, **INCREMENTS**, and **RELEASES**. **AGILE** emphasizes the concept of **DONE IS DONE**, where all activities and documentation for a piece of functionality are completed as new functionality is delivered, leaving little or no work to be cleaned up later. **AGILE** teams define "**DONE**ness criteria" that address all activities related to definition, implementation, and testing.

To align **AGILE** with regulatory expectations of a quality management system, **DONE**ness criteria should also address the following:

- The process activities that must be completed and, therefore, the process requirements that must be satisfied.
- The reviews to be performed and the approvals to be obtained (whether these are peer-review **VERIFICATION** activities or formal design reviews required by regulatory requirements).
- The product acceptance criteria used to evaluate the work product.
- The documentation to be produced (including the work products and the audit trail).

Define **DONE***ness to include the requirements of the quality management system that must be satisfied.*

5.1.9 Feedback mechanisms

When a large development effort is decomposed into small pieces of **STORY** work, it is important to consider whether anything that was once thought to be **DONE** has become un**DONE**. Impacts to consider could be upstream impacts (a change to implementation might suggest a change to requirements that were previously thought to be **DONE**), downstream impacts (a change in requirements might suggest a change to implementation that was previously thought to be **DONE**), or cross-system (a change in one feature's requirements or implementation might suggest a change to some other feature's requirements or implementation).

The integration activities performed within the **INCREMENT** and **RELEASE LAYER**S also provide feedback to demonstrate that the entire system has come together properly. These activities demonstrate that the sum- of-the-parts of the **STORY** work has been completed properly and that there were no unintended impacts on other parts of the system.

As the software emerges, activities or process outputs that were declared **DONE** in an early **INCREMENT** might become un**DONE** when subsequent **INCREMENT**S make changes. Software development plans must allow time for the necessary re-work loops and not allow schedule pressure to encourage the team to skip steps that are necessary to ensure that process outputs remain in a consistent state of **DONE**ness.

Establish feedback mechanisms that will identify whether re-work is necessary.

Practices such as continuous integration, automated regression testing, customer demos, and mid-level and high-level design reviews are all important in providing feedback that is essential to an **INCREMENTAL/EVOLUTIONARY** life cycle.

5.1.10 Aligning AGILE and non-AGILE development teams

Some development organizations use **AGILE** in part of the development process, but not for all activities related to product development. For example, on a large development effort with multiple teams, some development teams might use **AGILE** while other teams might not, such as a software development team using **AGILE**, while a hardware development team does not. Or the product development organization might use **AGILE**, but other organizations that provide input to product development or use the output of product development do not.

When different teams are using different methods, the following topics should be addressed to ensure that **AGILE** practices are used effectively, and that design control procedures are satisfied.

5.1.10.1 Planning and synchronization

AGILE'S INCREMENTAL/EVOLUTIONARY life cycle demands frequent planning to determine what development activities will be performed and what deliverables will be produced by what teams at what times. Whether this planning is **ITERATION** Planning or **INCREMENT** Planning in the time-boxed **AGILE** methods such as Scrum and SAFe, or frequent planning activities to manage Work In Process (WIP) in Kanban, these frequent planning activities performed by **AGILE** teams might not align with the less frequent planning activities of other non-**AGILE** development teams.

The requirement in 21CFR820.30(b) that states: "The plans shall identify and describe the interfaces with different groups or activities…" is particularly important in this context. As with any development effort that involves many teams, it is important to effectively identify responsibilities to ensure that activities are executed properly and that important activities are not missed. This is especially important with a development effort involving many teams who plan and execute differently. The following items should be considered when planning a project that includes **AGILE** and non-**AGILE** teams that operate on different planning cadences and with different planning mechanisms.

- A high-level project plan should describe the organization of the teams and identify the different planning cadences and different planning mechanisms used by the teams.
- A high-level project plan should describe how dependencies are managed and identify the synchronization points where all teams align on their plans, such as the identification of common milestones that all teams must align to (e.g., a business stage-gate review), or common events that require participation from all teams (e.g., a program-wide Design Review or a system integration activity).
- A high-level project plan and/or each team's specific plans should identify how each team will contribute content to shared deliverables, and how those deliverables will be approved.

Establish plans that demonstrate how all teams plan and execute effectively together, even though **AGILE** *and non-*AGILE *teams might plan and execute differently.*

Each team's plans should specify when they need participation from other teams in their planning activities, or in the execution of activities identified in their plans.

5.1.10.2 Aligning design inputs

While all teams need to have Design Inputs in order to produce Design Outputs, **AGILE** teams are more tolerant of a fluid and context-dependent definition of what it means to have Design Inputs before working on Design Outputs. (See Clause 5.3 on Design Inputs and Design Outputs for more on this topic.) **AGILE** teams address this in the concept of "Definition of Ready", where the **AGILE** team must determine that a **BACKLOG** Item is ready to be planned and delivered, and one criteria in the "Definition of Ready" is that the Design Inputs (requirements) are defined "enough" to begin.

In contrast, non-**AGILE** teams are often more strict in their expectations of what it means to have Design Inputs before working on Design Outputs. First, non-**AGILE** teams might require a specific sequence of activities, such as requiring that Design Inputs be documented and approved before starting work on a Design Output. Second, non-**AGILE** teams might be less tolerant of changing requirements, and therefore have more controls on changes to Design Inputs after they have been approved. Third, **AGILE** teams may have a different structure and granularity of Design Input compared to non-**AGILE** teams.

The following items should be considered when teams have different expectations and mechanisms for creating and changing Design Inputs.

A high-level project plan should describe (or reference to procedures that describe) the different structures, granularity, and mechanisms used by the teams to manage their Design Inputs.

A high-level project plan should describe the synchronization mechanisms and milestones to align the Design Inputs being managed by different teams.

The overall program should establish the organization of Design Inputs and how they are used by different teams, identifying those that can be completely managed by one team as they see fit, and those that must be managed such that the needs of more than one team are taken into account.

If a non-**AGILE** teams wants to complete their Design Inputs sooner than an **AGILE** team would complete theirs, but the non-**AGILE** team needs Design Inputs from an **AGILE** team in order to complete their Design Inputs, then the **AGILE** team might have to reprioritize their **BACKLOG** and work on completing some of their Design Inputs sooner than they might otherwise choose.

Each team should understand the mechanism each is using to manage changes to Design Inputs and participate in them appropriately. For example, if a non-**AGILE** team uses a Change Control Board (CCB) to manage changes after Design Inputs have been approved, then an **AGILE** team must know how and when they need to participate in the CCB. Likewise, if an **AGILE** team manages changes to their Design Inputs as part of managing the **BACKLOG**, then a non-**AGILE** team must know how and when to participate in managing the **BACKLOG**.

Establish the different mechanisms that teams might use to manage Design Inputs, emphasizing the synchronization and alignment elements to ensure Design Inputs are complete and consistent.

While **AGILE** teams are generally more tolerant of change because their product and processes are optimized to reduce the cost of change, they should recognize that other teams might not be as tolerant of change because their cost of change is higher. As the **BACKLOG** is created and refined, the system-level impact of potential changes should be considered in determining the order of the **BACKLOG**. When a change is to be made, the plan should consider the impact on the entire system and every team.

5.1.10.3 Aligning design outputs

When multiples teams are creating individual elements that are a part of broader system development, the product and associated deliverables might be completed in differing cadences. One difference could be that an **AGILE** team might deliver smaller bits of the product and small portions of the deliverables in a shorter cadence, whereas a non-**AGILE** team might deliver larger bits on a longer cadence. Another difference might not be related to **AGILE** at all, instead related to the nature of the product being developed.

In any event, the following items should be considered when different teams are on a different development cadence of producing their Design Outputs.

- A high-level project plan should describe the organization of the teams and the Design Outputs they are responsible for delivering, emphasizing which Design Outputs can be created independently of another team, and which require interactions between teams.
- Strategies for VERIFICATION and VALIDATION should establish the types of VERIFICATION and VALIDATION (V&V) activities to be preformed at what levels of the system, and establish which teams are responsible for those activities.
- When teams are separately developing components of a system that must be integrated together, a high-level project plan and/or team-specific plans should establish which team is responsible for the integration.

— When teams are developing a system that requires some VERIFICATION and VALIDATION activities that can be performed only on a complete, integrated system, a high-level project plan should establish the synchronization points where those V&V activities can be performed.

Establish synchronization points where the Design Outputs coming from many teams are integrated, verified, and validated together.

5.2 Inputs and outputs

The dynamic and varied way in which process activities can be executed adds complexity to the way in which process inputs are used and process outputs are generated.

In some linear manufacturing-line models, an activity produces a completed output, which is used in a subsequent activity that expects a completed input, which then generates its own completed output, and so on. Although this is a simple model, it is rarely practical or desirable for software development. It is unreasonable, for example, for a development team to complete and approve all requirements before any thought is given to design and test. **AGILE'S INCREMENTAL/EVOLUTIONARY** model acknowledges this and provides mechanisms that are both practical and desirable.

Every activity in a software development process must define the output that it produces and the inputs that are needed to produce that output. For every activity, the following criteria must be applied:

— What are the inputs?

— What are the entry criteria for those inputs? How **DONE** do they need to be before an activity can begin?

— What are the exit criteria? How **DONE** do the inputs need to be before the activity can be declared **DONE**? How **DONE** do the outputs need to bebefore the activity can be declared **DONE**?

Each of these criteria is addressed in the following sections.

5.2.1 Which inputs

AGILE'S **INCREMENTAL** nature emphasizes that only the inputs that are relevant to the activity need to be considered, so it is important to understand the scope of the activity before determining which inputs are to be considered.

Within the lowest layer of a **STORY**, which is small and tightly bounded, only the inputs relevant to the small and tightly bounded activities for that **STORY** are necessary. A complete set of inputs for all **STORIES** is not necessary. For example, only the requirements related to the **STORY** are to be considered, not all requirements for the entire system. Of course, the challenge is to determine what is relevant and what is not, so it is important to define the set of **STORIES** on the **BACKLOG** in a way that helps the development team make the determination. A useful criterion for each **STORY** is: "Have we identified the inputs that are relevant to this **STORY**?"

Similarly, within the **INCREMENT LAYER** only the inputs relevant to the scope of the **INCREMENT** are relevant, and so on to the **RELEASE** and **PRODUCT LAYERS**.

*Identify the inputs that are relevant to the work to be **DONE**.*

5.2.2 Entry criteria for inputs

AGILE'S **EVOLUTIONARY** nature allows for inputs to evolve as the outputs are created, so a good entry criterion is: "The input is **DONE** enough to allow us to proceed with this activity." This is a very subjective criterion, and it depends heavily on the context in which it applies and on the team's judgment in applying it. In some cases, it might be best for an input to be documented and approved. This is useful when there is a fundamental decision to be made on a direction to take or when there is significant risk in proceeding without an approved input. In other cases, it might be best for an input to only be conceptual, neither documented nor approved. This is useful when there are many input options and when some work could provide better clarity on the best option to pursue.

As is typical with decisions like "how **DONE** is **DONE** enough," such decisions are best made when balancing risk: the risk of locking down an input too soon versus the risk of locking it down too late. Applying the **LEAN** principle of "last responsible moment," decisions should be made as late as practical to allow the best solutions to emerge, but as soon as necessary to provide sufficient direction to the development team, to avoid unnecessary rework or to mitigate risk.

AGILE's planning mechanisms support this risk-based decision-making (see Appendix B). When the development team (which includes testers, coders, architects, and a customer representative) prioritizes a **STORY** to be worked in an upcoming **ITERATION**, two important criteria are the following:

— Do we know enough about this **STORY**'s inputs to proceed?

— Are we willing to let the inputs emerge as the **STORY** is worked?

Similarly, within the **INCREMENT**, **RELEASE**, and **PRODUCT LAYERS**, the criteria are "Do we know enough about this **INCREMENT'S/RELEASE**/product inputs to proceed?" and "Are we willing to let the inputs emerge as the **INCREMENT/RELEASE**/product is worked?"

*Inputs must be "**DONE** enough" to begin the work.*

5.2.3 Exit criteria for outputs

Regardless of the chosen life cycle model and the sequence of activities, it is important to ensure that inputs and outputs maintain a consistent state. (See a discussion of this issue in IEC 62304, Section B.1.1). **AGILE**'s **INCREMENTAL** approach emphasizes work on small pieces of functionality (a **STORY**) that is meant to be completed within tight **ITERATION** boundaries. To maintain the consistent state of inputs and outputs, the simplest and most ideal exit criteria for a **STORY** are the following:

— The **STORY** is not **DONE** until all the outputs and related inputs are **DONE**.

— **DONE** includes documentation and approval.

— All process outputs related to the **STORY** are consistent with one another.

*Inputs and outputs must be "**DONE**" to call the work **DONE**.*

To satisfy these criteria within the **ITERATION** boundaries, **AGILE** teams employ two strategies. First is to define a **STORY** to be small enough in scope such that the tasks to complete inputs and outputs are short enough to complete within the bounds of the **ITERATION**. Second is to identify potential delays (such as waiting for an approval) and task sequencing (such as completing a draft document, reviewing the document, making updates, and re-reviewing and approving the document) so that a plan can be made to complete all tasks within the bounds of the **ITERATION.**

Though this seems ideal, it is not always possible or desirable to satisfy these criteria for every **STORY**. It may be desirable in some cases to define a **STORY** that adds some piece of functionality that results in some temporary inconsistencies between some process outputs. When this happens, a new **STORY** should be created to address the completion of items necessary to make the system and the process outputs consistent again.

5.3 Design inputs and design outputs

The above discussion of inputs and outputs addresses the general concept of inputs to a process activity. Of particular interest in the medical device software world is the specific topic of **DESIGN INPUTS** and **DESIGN OUTPUTS**. When considering a complex medical device that consists of many subsystems and components, some of which could be software, the seemingly simple concept of **DESIGN INPUTS** and **DESIGN OUTPUTS** can become complex. For example, there can be high-level user needs that are inputs to system definition, which leads to system-level requirements, which in turn are allocated through the system design process to subsystems that include software, which results in software requirements that are input to software design, and so on.

In the discussion of **DESIGN INPUTS** and **DESIGN OUTPUTS** in this section, "**DESIGN INPUTS**" can be considered to be software-level user needs and intended uses (such as marketing requirements or human factors inputs) that define what the software must do and detailed requirements (such as functional specifications, performance requirements, or safety requirements) that establish the criteria that the software design must satisfy. "**DESIGN OUTPUTS**" can be considered to be the software product itself, along with the code that was compiled into it, the design documentation that describes it, and the test procedures and test results that demonstrate its correctness.

While the **AGILE** perspective does not typically use the terms "**DESIGN INPUT**" and "**DESIGN OUTPUT**," both concepts are well represented in **AGILE**. **DESIGN INPUTS** are represented by **STORIES**, which are written from a customer's perspective and which contain user needs and intended use; acceptance criteria on a **STORY**, which contain functional and non-functional requirements); and acceptance tests, which contain detailed requirements that the software design must satisfy. **DESIGN OUTPUTS** are represented by the design artifacts that are created to demonstrate that a **STORY** has been completely implemented and that all acceptance criteria have been satisfied.

Both the **AGILE** and regulatory perspectives align well on the value and importance of **DESIGN INPUT**S as the basis for the design of the software. Both align on the value and importance of **DESIGN OUTPUT**S that define the result of the design process in ways that allow the design to be validated. And, finally, both align on the value of **VERIFICATION** and **VALIDATION** to demonstrate that the **DESIGN OUTPUT**s align with the **DESIGN INPUT**S, that is, that the requirements, and ultimately the user's expectations, have been satisfied. For more information on the relationship of stories and requirements, see Sections 6.1.2.1, 6.1.2.2 and also Annex B.

AGILE brings a set of principles and practices for producing **DESIGN INPUT**S and **DESIGN OUTPUT**S that could be applied in ways that do not align with regulatory expectations. By following the recommendations in the following sections, designers can avoid those misalignments. It is important for software development plans and development processes to demonstrate that regulatory expectations for design controls have been applied to **DESIGN INPUT**S and **DESIGN OUTPUT**S.

*Show how definition activities satisfy regulatory expectations for **DESIGN INPUT**S, and how design activities satisfy regulatory expectations for **DESIGN OUTPUT**S*

5.3.1 Activities for producing DESIGN INPUTS and DESIGN OUTPUTS

Regulations such as 21 CFR 820.30 (Design Controls) and standards such as IEC 62304 provide requirements and expectations for the activities of producing **DESIGN INPUT**S and **DESIGN OUTPUT**S. The intent is to provide controls on the design process that ensure that high-quality software is produced. These controls can be readily applied to **AGILE** practices for delivering software within the **STORY**, **INCREMENT**, and **RELEASE LAYER**S.

Neither 21 CFR 820.30 nor IEC 62304 provides explicit requirements for the sequence of activities related to producing **DESIGN INPUTS** and **DESIGN OUTPUTS**, although they are sometimes interpreted as if they imply a linear life cycle. **AGILE'S INCREMENTAL/EVOLUTIONARY** life cycle describes the **DESIGN INPUT** and **DESIGN OUTPUT** activities as occurring in a more complex flow.

Regardless of the chosen life cycle, it is reasonable to think of **DESIGN INPUT** activities as occurring before **DESIGN OUTPUT** activities, as shown in Figure 6. Regulatory guidance and standards describe activities with an implication of an expected sequence.

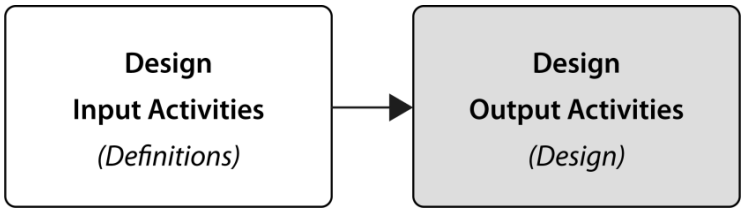

Figure 6—DESIGN INPUT/OUTPUT relationship: Highest level of abstraction

While the model shown in Figure 6 makes sense at a high level of abstraction, this model is too simple for the next level of detailed activities for producing **DESIGN INPUT**s and **DESIGN OUTPUT**s. It can be misleading or even incorrect to conclude that the model of Figure 6 means that all **DESIGN INPUT** activities must occur before any **DESIGN OUTPUT** activities can occur or that all **DESIGN INPUT**s must be completed and approved before any **DESIGN OUTPUT**s can be created.

A more realistic model must take into account the practical implication of the following topics, each of which is explored in the following sections:

- When working on small pieces of the software at a time, how do the input/output sequencing relationships work?
- How **DONE** do the high-level **DESIGN INPUT**s need to be for the activities that produce **DESIGN OUTPUT**s to begin?
- When do high-level **DESIGN INPUT**s get refined into detailed software requirements, and how does that transformation align with the activities that produce **DESIGN OUTPUT**s?
- What synchronization activities are necessary to ensure that **DESIGN OUTPUT**s satisfy the **DESIGN INPUT**s?

AGILE addresses these topics. This TIR provides guidance on how to address them in ways that satisfy regulatory expectations.

*Regardless of how the activities are organized, the activities for producing **DESIGN INPUTS** and **DESIGN OUTPUTS** must satisfy the expectations for design controls*

5.3.2 Breaking up the work

For all but the smallest software projects, the work performed on **DESIGN INPUT**s and **DESIGN OUTPUT**s is not performed in the large chunks shown in Figure 6. Instead, work is broken into small pieces, as shown in Figure 7.

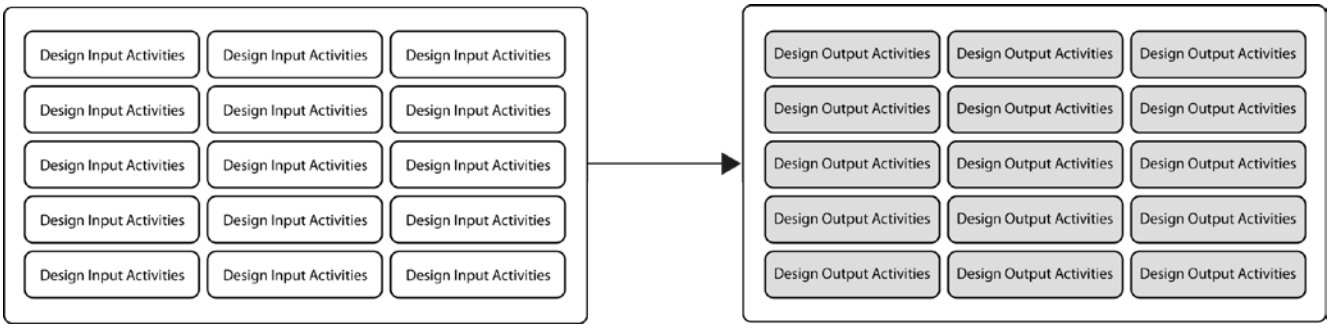

Figure 7—DESIGN INPUT/OUTPUT relationship: WATERFALL development

INCREMENTAL/EVOLUTIONARY life cycle models take this organization one step further by arranging the related **DESIGN INPUT** and **DESIGN OUTPUT** activities to be performed together. Rather than creating all the **DESIGN INPUT**s and then moving on to creating all the **DESIGN OUTPUT**s (see Figure 7), they are paired closely together in time, as shown in Figure 8.

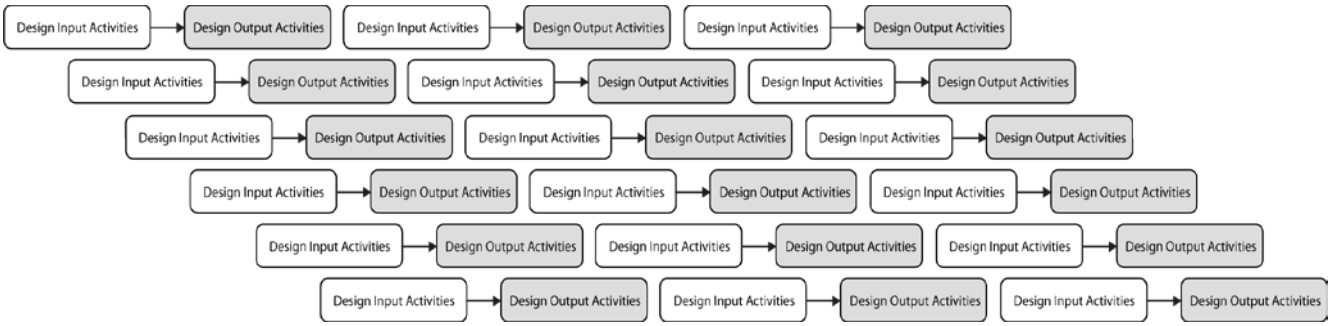

Figure 8—DESIGN INPUT/OUTPUT relationship: INCREMENTAL/EVOLUTIONARY

In this model, each small **DESIGN INPUT** precedes its corresponding **DESIGN OUTPUT**, but notice that all **DESIGN INPUT**S do not come before all **DESIGN OUTPUT**S.

*The relationship of **DESIGN INPUT**S TO **DESIGN OUTPUT**S IN AN **INCREMENTAL/EVOLUTIONARY** life cycle is best considered in the context of small pieces.*

The model shown in Figure 8 is certainly more complex than that shown in Figure 6, which puts a greater burden on planning and change management systems. In this more complex model, it is important to break up the work into separable and manageable pieces. As **DESIGN INPUT**S and **DESIGN OUTPUT**S are produced INCREMENTALLY, each piece should be properly reviewed and controlled.

Define the level of review and control that is applied to the **INCREMENTAL** creation of ***DESIGN INPUT**S and **DESIGN OUTPUT**S***

(See also the following section on synchronization for information on review and control at **INCREMENT** and **RELEASE** boundaries.)

5.3.3 Timing of design inputs and design outputs within an AGILE STORY

AGILE takes this organization even further, encouraging **DESIGN INPUT** activities to overlap with **DESIGN OUTPUT** activities. As described in Figure 9, a single **STORY** would encompass both **DESIGN INPUT** and **DESIGN OUTPUT** activities for a small piece of functionality.

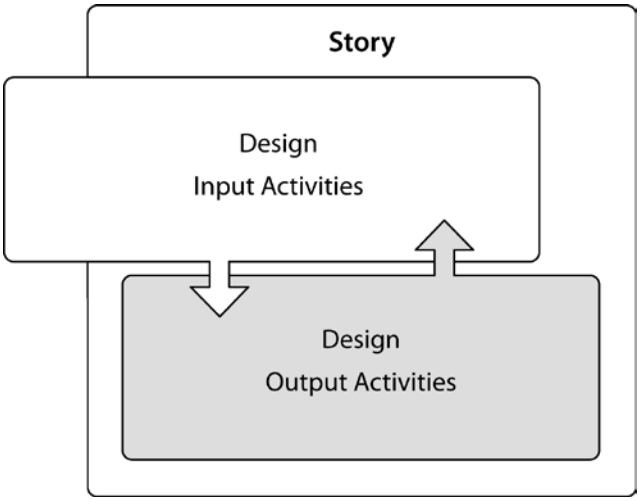

Figure 9—DESIGN INPUT/OUTPUT relationship: STORY level

As described by the bi-directional arrows in Figure 9, **AGILE** recognizes that **DESIGN INPUT**S influence the **DESIGN OUTPUT**S, but also that **DESIGN OUTPUT**S influence the **DESIGN INPUT**S.

DESIGN INPUTS clearly influence **DESIGN OUTPUT**S in that there must be an idea about the requirements of the problem to be solved before solutions can be implemented. It should be equally clear that **DESIGN OUTPUT**S can influence **DESIGN INPUTS** when solutions provide information on which requirement options are possible.

The challenge to an **AGILE** team is to manage the dynamic relationship of **DESIGN INPUT** and **DESIGN OUTPUT** activities that overlap.

Consider the more detailed activities that occur when creating **DESIGN INPUT**S and **DESIGN OUTPUT**S, as shown in Figure 10.

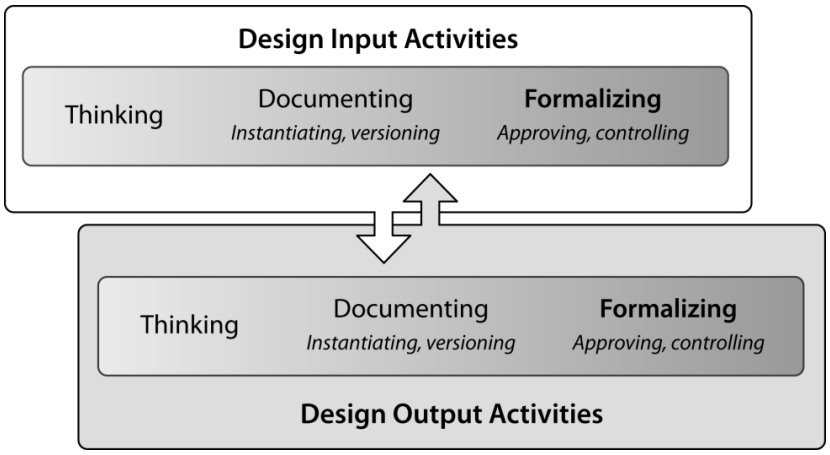

Figure 10—DESIGN INPUT/OUTPUT relationship: STORY level showing activities

The first step is thinking, in which ideas are being considered without the constraints of formal documentation and control. Thinking is intangible, although it might be conveyed in words and images or even as ideas drawn on a whiteboard. The second step is documenting, in which ideas are put into some form that makes them more easily and precisely conveyed to others. Documenting brings useful constraints and stability to an activity by forcing ideas to be formulated in a way that can be readily communicated and by forcing decisions on what is worth documenting. The third step is formalizing, an integral part of a formal, well- controlled quality system. Here is where documented ideas are declared "**DONE**," and some level of control and approval is applied. Formalizing adds yet another level of stability to the activity because it forces decisions on what is worth formalizing.

So, when we have expectations that "**DESIGN INPUT**S come before **DESIGN OUTPUT**S," what exactly do we mean? Consider Figure 11.

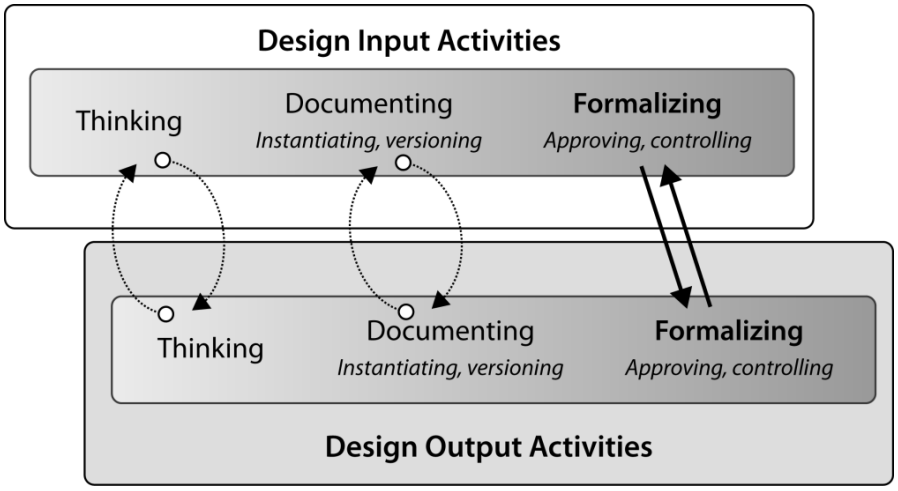

Figure 11—DESIGN INPUT/OUTPUT relationship: STORY level showing detail and sequencing

The flow of information shown in Figure 11 is more realistic than that shown in Figure 6, and although it is more complex, it can still satisfy the expectations for reasonable sequencing of activities.

First, notice there is no flow from "formalizing" of **DESIGN INPUTS** to the "thinking" of **DESIGN INPUTS**. When we say that "**DESIGN INPUTS** should precede **DESIGN OUTPUT**S," it is unreasonable to mean that the inputs must be formalized before we think about **DESIGN OUTPUTS**. In fact, it is usually desirable to give some thought to **DESIGN OUTPUT**S in order to be comfortable that **DESIGN INPUTS** can be formalized.

The circular arrow connecting the two "thinking" steps reflects the dynamic nature of software definition and design. Ideas about what the software should do will influence the design of how to do it, and ideas about the design solutions can also influence the ideas for definition. There is very little constraint on the order in which the development team thinks, so this loop could be thought of as one continuous activity.

Similarly, the circular arrow connecting the two "documenting" activities shows that it is reasonable to document **DESIGN INPUT**S and **DESIGN OUTPUT**S in a dynamic sequence, depending on what is worth documenting first and how one influences the other. Because documenting requires some cost and because it will narrow the number of ideas being considered, the number of documentation loops and the best sequence of activity might be driven by cost considerations and by the decisions made on which ideas to pursue.

The loops of thinking about and documenting the **DESIGN INPUT**S and **DESIGN OUTPUT**S can be thought of as being in a draft stage.

Moving to the formalization step signifies moving from a draft stage into a more controlled stage of **DONE**ness. The sequence of formalizing **DESIGN INPUT**S and **DESIGN OUTPUT**S is not the same loop as the other steps. Formalization is driven by the desire to bring more control to the design process and to finalize definition and design decisions, so there is significantly less looping between the formalization activities. Also, depending on what decisions are being made and how they are being made, there is often a specific sequence that matters. This relationship is shown as a bi-directional arrow to show that any sequence is viable.

In some cases, the sequence is to formalize the **DESIGN INPUT**S before formalizing the **DESIGN OUTPUT**S. This is the typical expectation, which is aligned with the broad expectation shown in Figure 6. It makes sense when there is value in ensuring that the **DESIGN INPUT**S are complete and approved before spending effort to complete and approve the **DESIGN OUTPUT**S.

The reverse, where **DESIGN OUTPUT**S are formalized before the **DESIGN INPUT**S are formalized, might not be as typical. Although this sequence might seem to conflict with the expectation of Figure 6, it can be reasonable. This sequence makes sense when it is necessary to evaluate the design solution represented by the **DESIGN OUTPUT**S before the **DESIGN INPUT**S can be declared good and **DONE**. For this sequence to work, it is necessary to know what state the **DESIGN INPUT**S were in when the **DESIGN OUTPUT**S were formalized, so that the formalization of the **DESIGN INPUT**S can demonstrate that they are consistent with the completed **DESIGN OUTPUT**S. The nature of the work being **DONE** is further described in Clause 6.1.2.2, "Requirements **DONE** when a **STORY** is **DONE**."

To manage the dynamic sequencing described here, **AGILE** teams rely on robust planning practices with continuous attention to the following question: "Do we have the right sequence of activities?" Of particular interest to regulators are the mechanisms and timing of the control and approval activities.

*Manage the dynamic timing and sequencing of **DESIGN INPUT** and **DESIGN OUTPUT** activities with robust planning practices*

5.3.4 Inputs to AGILE stories

Note that in Figure 9 the boundary of a **STORY** does not entirely encompass the **DESIGN INPUT** activity. As a large development effort is broken into **STORIES**, some definition must be worked on when creating the **STORY** to put on the **BACKLOG**, and some are worked on when the **STORY** is being worked, in parallel with the work on the **DESIGN OUTPUT**S. The degree of overlap depends on factors unique to a specific **STORY**.

On one extreme, the **DESIGN INPUT** definition performed outside the **STORY** might be nothing more than a vague understanding of preliminary user needs, only enough to understand the meaning and scope of the **STORY**. In this

case, most of the definition work is performed inside the **STORY** to further refine the user needs and define the detailed requirements. This definition relies on a strong customer role in establishing requirements and acceptance criteria as the development team is creating the solution. This scenario is reasonable when customers are not clear on what the requirements should be until they see more of the solution possibilities or when the exploring of solutions might uncover requirements that were not easy to see upfront.

On the other extreme, the **DESIGN INPUT** definition performed outside the **STORY** might be a complete definition of user needs and final requirements. The work inside the **STORY** would then be nothing more than confirming that those requirements are reasonable by completing the **DESIGN OUTPUT**s to satisfy them. This scenario is reasonable when customers are very clear on what they want or when the solutions are well defined and would not benefit from exploring options.

On either extreme or anywhere in between, by the time the **STORY** is completed, the **DESIGN INPUT**s must be complete and consistent with the **DESIGN OUTPUTS** that they drive.

*The degree to which **DESIGN INPUT** precedes **DESIGN OUTPUT** depends on the unique factors of a particular **STORY**.*

5.3.5 Synchronizing design inputs and design outputs

In contrast to the simple relationship of **DESIGN INPUT**s and **DESIGN OUTPUT**s shown in Figure 6, the **AGILE** model is more complex, as shown in Figure 12.

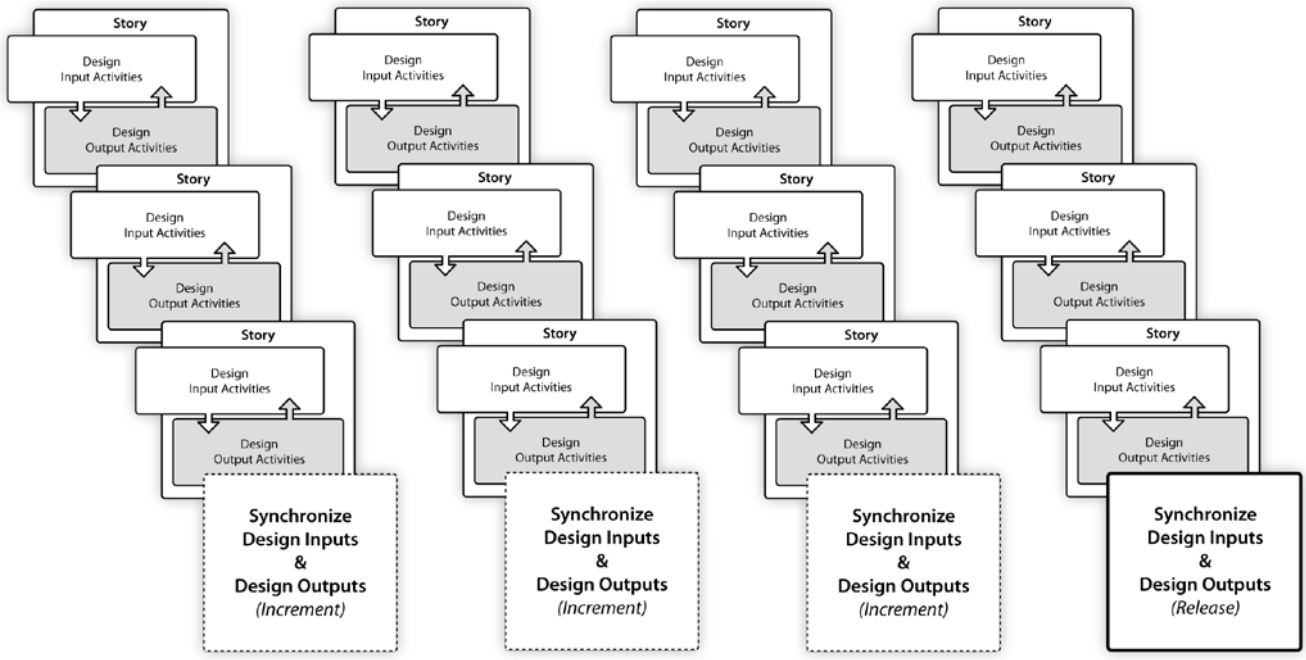

Figure 12—Synchronizing DESIGN INPUT/OUTPUT at INCREMENT and RELEASE boundaries

A concern with this highly **INCREMENTAL/EVOLUTIONARY** life cycle and the ideas discussed above is that the **DESIGN INPUT**s and **DESIGN OUTPUT**s could become inconsistent or in conflict with each other as the pieces are developed. To manage that concern, **AGILE** uses concepts of **INCREMENT**s and **RELEASE**s to synchronize the **DESIGN INPUT**s and **DESIGN OUTPUT**s.

At an **INCREMENT** boundary, the synchronization includes a review of the **DESIGN INPUT**s and **DESIGN OUTPUT**s, both individually and as a set. There might be a review of the complete set of **DESIGN INPUT**s to evaluate the completeness of the set, assess whether the work performed on the definition of individual **STORIES** resulted in a definition that is consistent and free of conflicts, and establish expectations for the next **INCREMENT** on whether definition is complete

enough to move on or issues need to be revisited. Similarly, there might be a review of the **DESIGN OUTPUT**s to evaluate completeness, assess consistency, and establish expectations for future work.

The synchronization also includes activities to determine whether the complete set of **DESIGN OUTPUT**s is consistent with and satisfies the **DESIGN INPUT**s. This might be a review, or it might be the execution of regression tests.

At **INCREMENT** boundaries, portions of the **DESIGN OUTPUT**s might be considered "**DONE**," but the entire set would not be **DONE**, because future work is planned for later **INCREMENT**s.

At the **RELEASE** boundary, the same kind of synchronization activities will occur as at an **INCREMENT** boundary, but with a higher expectation for **DONE**ness on the entire set. Whereas at an **INCREMENT** boundary it is expected that some portions of the **DESIGN INPUT**s and **DESIGN OUTPUT**s will be incomplete, the expectation at a **RELEASE** boundary is that the entire set will be fully traceable and complete enough to satisfy the purpose of the **RELEASE**.

AGILE *projects should define at which synchronization points the reviews should happen and the level of review and control applied.*

See also the previous section on breaking up work for information on review and control as **DESIGN INPUT**s and **DESIGN OUTPUT**s are produced.

5.3.6 Final VERIFICATION

When software is ready for **RELEASE**, the final **VERIFICATION** activities must demonstrate that the final product satisfies its **DESIGN INPUT** requirements. With this constraint, it is reasonable to expect that **DESIGN INPUT**s are "**DONE**" (meaning "formalized" in the discussion above) before the final **VERIFICATION** is performed. This is one point at which it is reasonable to expect a "finish-to-start" relationship between **DESIGN INPUT** activities and final **VERIFICATION** activities.

A simple way to satisfy this expectation is to execute a complete **VERIFICATION** of the final software against the complete and final set of **DESIGN INPUT**s. **AGILE**'s principles of continuous **VERIFICATION** and automated regression testing encourage this approach, but still it is often not practical to perform every possible **VERIFICATION** activity on the final version of software.

It is tempting, therefore, to "take credit" for **VERIFICATION** performed at previous **INCREMENT** boundaries and during **STORY** development. This is a reasonable approach, but only if strong change management mechanisms (e.g., software ripple analysis) are used to ensure that ***VERIFICATION*** activities performed for previous versions of the software against earlier versions of **DESIGN INPUT**s were not invalidated by changes made later.

*Perform final VERIFICATION against final **DESIGN INPUT**s, while considering the **VERIFICATION** DONE in earlier work.*

5.4 Design reviews

Regulations and standards set expectations for reviews to take place throughout the software development process. Comprehensive reviews such as those described in 21 CFR 820 Subpart C are performed at stage boundaries. Smaller review activities, such as technical reviews and document reviews, are performed as activities are completed and documents are produced.

In an **AGILE** model, reviews occur at multiple points in the software development life cycle. To clearly map the reviews that take place in an **AGILE** life cycle to the reviews expected by regulations and guidance, software development plans should clearly describe the reviews that occur. Reviews that satisfy regulatory requirements should be documented.

In software development plans, define the reviews that take place, showing that they satisfy regulatory requirements.

5.4.1 Formal design review at stage boundaries

The FDA QSR and related guidance require formal design reviews at the completion of stages of development to demonstrate readiness to proceed with the next stage. These reviews are intended to be comprehensive for the stage being reviewed. A project's software development life cycle determines those stages.

In **AGILE**'S **INCREMENTAL**/**EVOLUTIONARY** life cycle model, stages are defined by the **INCREMENT** AND **RELEASE** boundaries, so formal design reviews can be performed as part of completing **INCREMENT**S and **RELEASE**S. Not all **INCREMENT**S will be broad enough in scope to represent a stage of the project, so software development plans should be clear on which boundaries require a formal design review.

Plan formal design reviews to be performed at **INCREMENT** *and* ***RELEASE*** *boundaries.*

5.4.2 Reviews as a **VERIFICATION** activity

Regulations and standards require that process activities be completed with a **VERIFICATION** activity that demonstrates that the process was completed correctly and that the resulting **DESIGN OUTPUT**S are complete and correct. This **VERIFICATION** is often performed through a review activity, during which reviewers evaluate the work products and the process used to complete them.

In an **AGILE** model, these reviews are often performed for each **STORY**, where a completion criterion for every **STORY** would include a review of the outputs of that **STORY**. To demonstrate successful completion of many interrelated **STORIES**, an additional review might be performed at **INCREMENT** boundaries.

Plan VERIFICATION review activities as part of ***STORY*** *completion and at* **INCREMENT** *boundaries.*

5.4.3 Independence of review

It is well established that effective reviews need some degree of independence of the reviewers, but the degree depends on the nature of the review. Regulations and standards are clear on expectations for independence during the formal design reviews performed at stage boundaries. For example, the FDA QSR, 21 CFR 820, Subpart C, element (e), states that design reviews shall include an individual(s) who does not have direct responsibility for the design stage under review.

All software development projects, whether using **AGILE** or not, should ensure that formal design reviews meet regulatory expectations by ensuring that an independent reviewer participates, and nothing about **AGILE**'S principles and practices should contradict that expectation.

Regulations and standards are not as clear on the expectation for independence during the smaller-scoped reviews that occur throughout a software development life cycle. Software development teams sometimes mistakenly apply the same expectations to the broader-scoped formal design reviews, which leads to perceived conflicts with some of the principles and practices that **AGILE** uses to address independence.

In **AGILE**, independence is manifested in several ways. The **AGILE** principle of collective ownership means that everyone on the project team is responsible for the quality of the software product, not just independent reviewers. **AGILE** practices such as pair programming, daily stand-ups, and direct involvement of the customer address independence by encouraging teams to bring different perspectives to their work throughout the development cycle. When planning the activities of work, teams decide which kinds of reviews are necessary and who should participate in them.

Regardless of the scope and purpose of a review, teams should consider whether reviewers have a sufficient degree of independence to ensure a good review.

Ensure objective review of work products by establishing an independent review channel for those who evaluate work products.

5.5 Documentation

A common but incorrect view of **AGILE** is that "**AGILE** says documentation is not necessary." The **AGILE** Manifesto's value statement of "working software over comprehensive documentation" can be misinterpreted, leading to this incorrect conclusion. For organizations or people who do not value documentation, they could embrace **AGILE** as a means to support their view. For organizations or people who value documentation, they could criticize **AGILE** as being incompatible for them.

A more reasonable view is to accept that documentation has differing value to different organizations, people, and contexts. Embracing this view, **AGILE** principles and practices can be used to ensure that documentation provides legitimate business value at a manageable cost.

Produce documentation that has business value.

Documentation clearly has legitimate business value for the medical device software world, both to internal **STAKEHOLDERS**, such as software development teams and project management, and to external **STAKEHOLDERS**, such as **CUSTOMERS** and regulators. **AGILE** should not be used as a means to undermine that value, although **AGILE** does bring some different views about documentation that might be challenging to the medical device software world. To address the challenge, some important points about documentation in an **AGILE** model are described here.

5.5.1 Use of documentation

Documentation is a means of communication. Its value is defined by the information it communicates, as determined by the people it is communicating between.

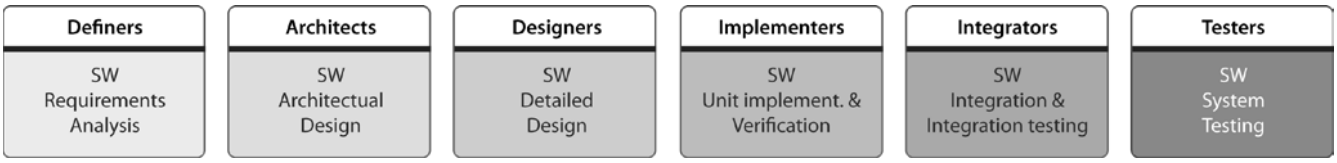

Figure 13—A linear flow of process activities

In a linear-flow model, where process activities are executed by different people over different times, documentation provides a means for those people to communicate over time. In the linear flow of activities shown in Figure 13, each activity would produce documentation to be handed off to the person who needs it in the subsequent activity. Definers produce requirements documentation to be handed off to architects to do architectural design, who then produce architecture documentation to be handed off to designers to do detailed design, and so on.

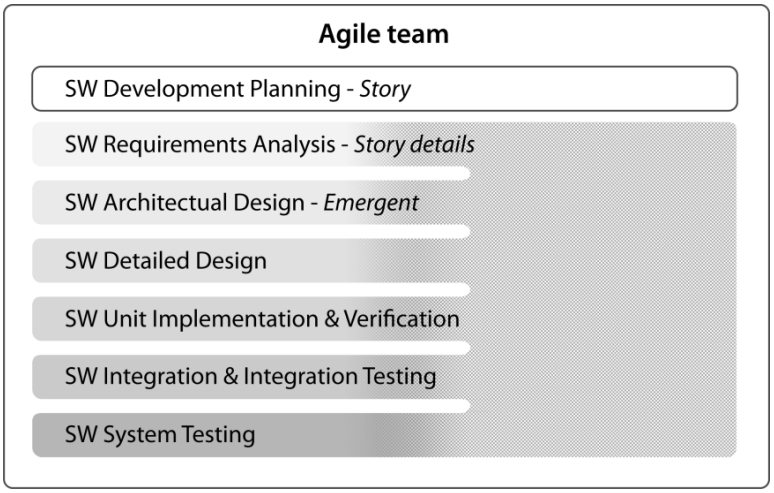

Figure 14—A parallel flow of process activities

In a parallel-flow model, the need for documentation changes significantly. To the extent that a single team consists of people playing multiple roles and working on multiple activities at the same time, there is less need and less value in producing written documentation as the main mechanism of communication between those people and activities. Instead, there can be more value in higher-bandwidth and more flexible means of communication, such as conversation, whiteboard diagrams, prototypes, visual models, and, ultimately, as the **AGILE** Manifesto emphasizes, the software itself. When information is needed to perform an activity, the team must always consider, "Do we have the information we need to begin and work on this activity, whether it is documented or not?"

Even in **AGILE**'S model of team-centered parallel activities, documentation maintains its importance in communicating the results of a team's activities. There are two important audiences to consider.

The first audience is the external STAKEHOLDERS, such as customers, regulators, and auditors, who need information about what was completed in order to assess the product and the processes used to create it. Considering the customer, the team must consider, "When we are DONE, will the customer have the information they need to assess what we've completed and be able to accept it?" Considering regulators and auditors, the team must consider, "When we are DONE, will we have sufficient evidence to demonstrate that our quality system processes were followed correctly?"

The second audience is a development team that will take the results of the team's activities to build upon in some future work. This audience is particularly important in an **AGILE** model. Because **STORIES** build upon the activities of previous **STORIES** (and **INCREMENT**S and **RELEASE**S), there is a continuous need to communicate the results of one set of activities so the next set can build upon them. To satisfy this need the team must consider, "When we are DONE with this **STORY**/**INCREMENT**/**RELEASE**, will we have the documentation that will be valuable to whoever works on the next **STORY**/**INCREMENT**/**RELEASE**?"

For either audience, documentation serves as the living "memory" of what was done. Although the high- bandwidth communication of a highly collaborative project team is effective as the team is working together, written documentation is effective for capturing information that must live on after the team completes its work. When determining what to document, the team should consider, "What will someone need to know later?" If the software is intended to be expanded later, future development teams would need to understand the software sufficiently to build upon it. If errors are discovered when the software is put into use, maintenance teams and regulators would need enough information to assess and correct the problem.

Produce documentation that communicates information to the intended audience.

5.5.2 Sequencing of documentation activities

In the previous section, notice the emphasis on "when we are DONE." In an **AGILE** model, where a team is working together on a set of activities, documentation is less important to initiating an activity ("when we begin") and guiding an activity ("while we are working"), but documentation is still important to communicating the results of the activity ("when we are DONE"). In **AGILE'S** highly iterative and **EVOLUTIONARY** model, which is executed by a team working on multiple process activities at the same time, it is more likely that documentation is a late or even final task in an activity, it is often not an early task. When a team is effectively collaborating with multiple communication methods, formal documentation often brings overhead that is not necessary or useful at an early stage. The creation of formal documentation is certainly a necessary element of defining that an activity is **DONE** but is often not necessary for determining whether an activity can commence.

When activities are done in parallel by a team working on both definition and design at the same time, the flow of information can take many paths. The information passed between activities could be thoughts, writing, or formal documents. In this model, both the question and the answers about **DONE**ness of inputs are different. Rather than asking the question, "Is the software definition **DONE?**" we instead ask, "Is the software definition **DONE** enough for what we are about to do for software design?" Many answers are possible to this question, such as "Yes, the thoughts on definition are **DONE** enough to explore some design options" or "Yes, the documentation is **DONE** enough for me to understand what I have to do to document the design" or "Yes, the definition is approved so I can finish the design with lower risk of definition changes causing rework."

For this to work, to produce a safe and effective product in a way that satisfies an auditor's expectations of a robust process, it is imperative that a team ask the questions described above to determine what level of documentation is needed to make progress on activities that need the content of that documentation as an input.

Produce documentation at the point in time when it fits the flow of creating it and using it.

5.5.3 Sum-of-the-parts of documentation

Because **AGILE'S INCREMENTAL/EVOLUTIONARY** life cycle encourages short activities to produce small bits of functionality at a time, a project's documentation will also be produced in small bits at a time.

As **STORIES** are being worked, the documentation associated with each process activity is changed, which means that any big document such as a requirements document, a test specification, or the code itself is almost always in some state of change. Therefore, there is less value and less relevance in asking "How **DONE** is the requirements document?" because the answer will depend on which parts of the document you choose to ask about. Only at **INCREMENT** or **RELEASE** boundaries, depending on the purpose of the **INCREMENT/RELEASE**, is it important to assess the state of the entire document.

In a process model such as IEC 62304, which defines activities to document and verify something, it is necessary to consider those activities at two levels, one being the small-part level where the work is DONE on the content and the other being the big-document level where the content comes together to create a consistent whole.

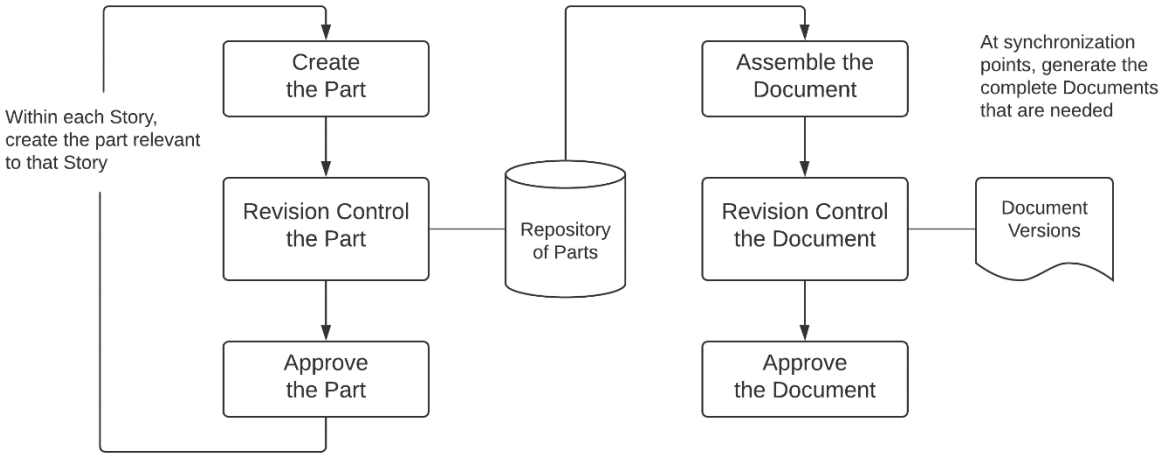

Figure 15—Sum-of-the-parts documentation

In an **AGILE** model, the small-part level is typically performed a **STORY** at a time, whereas the big-document level is typically performed as a task separate from a **STORY**, for reasons relevant to completing an **INCREMENT** or a **RELEASE**.

As shown in Figure 15, each **STORY** creates parts of documentation (individual requirements, portions of design documentation, small modules of code, individual tests, and test results, etc.), puts those parts in revision control, and approves those parts. At an **INCREMENT** or **RELEASE** when an aggregate document is needed, that document is assembled from the parts that were revision controlled, and that aggregate document is put in revision control and approved.

Define how documentation is written, controlled, and approved as a sum-of-its-parts.

The parts of documentation should be written, controlled, and approved at the same time that the software itself is created, controlled, and approved. The sum-of-the-parts documentation should be created when there is need for a complete document. Documentation should not be an afterthought that is addressed only after the software is completed.

5.5.4 Process artifacts (the audit trail)

In addition to the work products (**DESIGN OUTPUTS**) that are generated as a result of a process activity, some activities produce a process artifact to demonstrate that the activity was performed in accordance with the requirements of the quality management system. Such artifacts (which could include meeting minutes, summary reports, test reports, or signoffs in a change management system) are necessary for regulatory STAKEHOLDERS who assess compliance with the quality management system, although they can also be useful to the development team in managing the design process.

Process artifacts (an audit trail) are necessary to satisfy regulatory expectations. In an **AGILE** model, where processes are executed many times in many **ITERATIONS** and levels of granularity, the cost of burdensome rules for process documentation is greatly magnified. The challenge to an **AGILE** team is to satisfy regulations for an audit trail while making the cost acceptable.

Regulations and standards can be interpreted as requiring activities to be performed in a specific sequence and, therefore, requiring audit-trail documentation to prove that the proper sequence was followed. For example, it can be interpreted that requirements documents must be signed off before design documents are signed off. In **AGILE'S INCREMENTAL/EVOLUTIONARY** life cycle, especially in the **STORY LAYER** where process activities are executed in varied and dynamic ways, it is often not useful to produce process artifacts to demonstrate the sequence in which activities

are performed. Instead, process artifacts demonstrate that an activity was completed properly, regardless of the sequence.

To reconcile the difference in the regulatory and **AGILE** perspectives for process documentation, plans in the **PRODUCT LAYER** and the **STORY LAYER** should set the expectations for the sequencing of activities as well as the process artifacts to be produced. To satisfy both perspectives, process artifacts should be a simple byproduct of executing a robust process.

Set clear expectations for the audit trail to be produced throughout the dynamic execution of an **AGILE** *project.*

5.5.5 Approvals and evidence

In **AGILE'S** highly INCREMENTAL/EVOLUTIONARY development model, there are many points in the development cycle where people do the following activities: (1) plan work to do (and the documentation to be created) (2) do work (and create documentation), (3) review that work (and the associated documentation), and (4) approve the completion of that work (and the associated documentation). Consistent with the **AGILE** principle of "Getting to DONE," each of these activities needs some level of agreement or approval that the activity has been performed correctly and sufficiently. For example, when planning the work to do, **AGILE** teams apply the practice of "Definition of Ready," where the **AGILE** team approves that the plan has been defined sufficiently so that they can proceed with execution. Further, the **AGILE** practice of "Definition of DONE" means that the team that did the work, reviewed the work, and approved the work all agree/approve that the work is DONE and documented correctly.

These approval controls within the INCREMENTAL model are important elements of Design Controls, and therefore relevant to regulators. With that, it is not enough to simply state that the **AGILE** practices of "Definition of Ready" and "Definition of DONE" are applied, evidence must be provided.

A challenge to **AGILE** teams is to provide sufficient evidence to demonstrate control, but not unduly burden the teams with documentation that provides little benefit to them. Of particular interest to **AGILE** teams is the question of whether a signature (either physical or electronic) is required to demonstrate approval.

From many years of experience with paper-based documentation where a signature on a document demonstrated approval, some organizations now assume that a signature is required to demonstrate approval. This is an assumption based on implementation choices, not on requirements from regulations or standards.

The FDA Quality System Regulation has several sections where approval is required for a document, such as the specific cases of some elements of 21 CFR 820.30 Design Controls Section, and the more general case of 21 CRF 820.40 Document Controls Section (a) which requires that Quality Management System (QMS) documents be approved before issuance. In most cases where approval is specifically required, there is a further statement that says, "The approval, including the date and signature of the individual(s) approving, shall be documented." Because the regulation so often states that specific approvals require a signature, it seems reasonable to link any requirement for an "approval" with an accompanying requirement for a "signature," but it is important to note that the regulation does not explicitly state that every kind of approval must include a signature as evidence.

ISO 13485 is much less specific. Section 4.2.4 Control of Documents is one of the few sections that mentions approval, and it does not specify the form of that approval, nor does it mention signatures. Similarly, IEC 62304 has many requirements for the approval of an activity and/or a document, but none of those requirements address the evidence to be provided and none mention the need for a signature.

With few exceptions, regulations do not provide specific requirements for approval evidence, leaving the manufacturer the freedom to determine the most appropriate way to demonstrate approval. The form and location of the evidence will depend on the tools used to manage design artifacts, the process rules established to use those tools, and the level of control the organization requires on approvals. Examples of approval evidence to consider are:

- The existence of design artifacts in their repository/tool (e.g., documents checked into the document control system, code checked into the source code management tool, tests created in a test management tool, etc.), when the act of putting an artifact into the repository is defined in procedures to be an approval action.

— An explicit approval indication on a specific design artifact, such as an approval-indication field in the document control system or a review/approval record associated with the check-in of the artifact

— An approval field(s) in a change-management record (or **BACKLOG** Item) indicating approval of all artifacts changed under that record

— A signature (either electronic or physical) linked to the design artifact.

When defining an activity that needs approval, specify the evidence to be provided.

Because the **AGILE** model demands frequent reviews and approvals, the mechanism for providing approval evidence should be fast and efficient, while still demonstrating sufficient control of the approval process.

Given that regulations and standards do not specify whether signatures are necessary to demonstrate any kind of approval, organizations should not simply assume that an approval must include a signature. Instead, when defining a process that includes an approval of an activity and/or a document, that process should explicitly state whether a signature is necessary.

When specifying the need to approve an activity and/or document, explicitly state whether the evidence of that approval must include a signature.

Beyond simply meeting the requirements of 21 CFR 820.40(a) for signatures, the definers of a process should consider the intent and value of having a signature, and then define their process accordingly. The following recommendations should be considered when defining the approval process and the need for signatures:

As shown in Figure 15, complete documents are generated when there is a need to aggregate a set of information that is developed incrementally via **STORIES**. When the purpose of that complete document is to aggregate the documentation relevant to the product to be released, a signature is useful to demonstrate the legal accountability that comes with the approval and to meet the requirement of 21 CFR 820.40(a).

For small changes to documents that are made and approved a **STORY** at a time (as shown in Figure 15), where the accountability is distributed across the **AGILE** team, a signature does not add value in demonstrating that accountability, so the process should explicitly state that a signature is not required. Regulations and standards do not require signature as evidence of approval at this layer.

When a document is created in INCREMENTAL steps whether the INCREMENTAL approval of that step would find value from having a signature might depend on the scope, purpose and STAKEHOLDERS' needs of the Increment. For example, if the INCREMENTAL approval is for an intermediate version of a document that will be updated and approved again before RELEASE, a signature may not be valuable for the evolutionary change, whereas if the document is created in INCREMENTAL pieces where the final version is simply an aggregate of those approved pieces, a signature might be useful on each piece to demonstrate the INCREMENTAL approval. Given the context-dependent decision on whether a signature is useful, the process for INCREMENTAL approvals should state that the Increment Plan will specify whether signatures are required.

While signatures are required by 21 CFR 820.40(a) for final document approvals, they are not required for intermediate approvals, and therefore you have flexibility in what your process should require.

5.6 Managing the dynamic nature of AGILE

INCREMENTAL/EVOLUTIONARY models such as **AGILE** are not as simple and tidy as the more linear **WATERFALL** model. Linear models have cleanly defined activities that are executed in simple sequences to produce complete artifacts that are simpler to document and demonstrate, whereas **INCREMENTAL/EVOLUTIONARY** models have those same activities, but they are executed in more dynamic and complex sequences to produce artifacts in varying stages of evolving completeness. An **INCREMENTAL/EVOLUTIONARY** model must rely on mechanisms to manage its inherent complexity. **AGILE** has many general concepts and specific practices that help manage the complexity and give feedback to the development team on the effectiveness of the life cycle and the evolution of the product.

5.6.1 Embrace change, manage change

The **AGILE** perspective emphasizes change as something to be embraced for competitive advantage, with practices that enable change in an effective way. **AGILE** recognizes that product definition will change over time, as customers' needs change or as innovation suggests new solutions. Likewise, product design will change over time as new solutions emerge. An **AGILE** project can be viewed as being in a constant state of change, from the point where the first decision is made on defining the product to the point where the final version of the product is shipped. In such a state, processes and practices are designed to apply to any change, whether it is developing a new system, adding new functionality to an existing system, or making a small change to correct an error in an existing system.

In a strictly linear **WATERFALL** life cycle, change management mechanisms focus on change control. Once a deliverable is approved, change management mechanisms apply restrictions that make change more difficult, so the process for making changes is thorough and robust, but not necessarily easy and sometimes even constraining. This can result in two sets of process rules: one set that applies to the initial creation of a deliverable and one set that applies to changes made to that deliverable.

In **AGILE'S** highly **INCREMENTAL**/**EVOLUTIONARY** life cycle, change management mechanisms focus on change facilitation. Deliverables are recognized as always being in a state-of-change, so the process for making changes is also thorough and robust, but in a way that facilitates frequent changes. This results in a single set of process rules that apply all the time for any change, albeit with some differences depending on the scope and nature of the change.

Frequent changes to software can introduce unintended side effects or defects into previously verified code, which presents a risk to software quality. Regulations, standards, and guidance address this risk by setting expectations for an effective change management system. They emphasize important mechanisms such as review and approval activities, impact assessment, **TRACEABILITY**, and change records. To align the regulatory expectations with the **AGILE** concepts, these change management mechanisms should be incorporated into the completion of **STORIES**, **INCREMENTS**, and **RELEASES**.

Effective change management mechanisms are essential to a highly **INCREMENTAL** life cycle.

5.6.2 Satisfy the customer

AGILE emphasizes the desire to produce software that satisfies the customer. It strives to involve the customer heavily throughout the design process so as to enhance the likelihood of satisfying the needs of the user and meeting the intended use of the medical device. Although publications from either perspective might use different words to reflect their particular emphasis, they align well on a common goal.

An active customer role is defined as a customer who works closely with the design team to define and design the software. By defining the product **BACKLOG** of **STORIES** that represent customer value to be delivered, by participating with the development team to design solutions, and by regular demos that validate the evolving software, the active customer role ensures that the system evolves in a way that meets the intended use.

Because there are many types of customers for medical device software, it is a challenge to define who the customer is and how that customer's role is played within an **AGILE** development team.

Continuously address customer needs through an active customer role.

5.6.3 Maintain the software development process

Both the **AGILE** and regulatory perspectives emphasize the need to tailor processes and practices to fit the context and to continually seek improvements. Although the goals of those improvements might differ, the principle of continuous improvement and the practices to support it are the same.

A common **AGILE** principle is "inspect and adapt," which means that teams are obligated to inspect both the software product they produce and the processes and plans they use to produce it, and then take action to adapt the product,

process, and plans depending on what they learn. This principle is useful for managing the complexity of executing a software development process in a highly **INCREMENTAL**/**EVOLUTIONARY** life cycle.

Through the practice of retrospections, in which teams ask, "Did we learn anything about our process from the **INCREMENT**/milestone we just completed?," and the practice of reflections, in which teams ask, "How is our process working for us?," **AGILE** teams are encouraged to optimize the processes they use. Through bringing in the regulatory perspective by asking "Is our process compliant with regulatory expectations?," teams can ensure that their development processes are effective and controlled.

*Establish effective **RETROSPECTIVE** and reflection practices to support continuous improvement and alignment with regulatory expectations.*

5.6.4 The role of software tools in regulated AGILE

AGILE teams may leverage a multitude of off-the shelf software tools to support their development work, enhance their agility, and automate their **AGILE** infrastructure. When operating in a regulated environment, it is important to determine whether the information captured via those tools is part of a regulated process and therefore subject to specific requirements.

In fact, medical device regulations stress the importance of maintaining appropriate documentation and records. Among other things, good quality records are necessary to demonstrate compliance with regulatory requirements, to maintain history of development activities and to drive process improvements. Some documents and records are explicitly mandated by regulations and standards, while others are implicitly needed to demonstrate compliance. In addition, good quality records are necessary to maintain a history of development activities, reproduce old RELEASES when needed, support problem resolution processes, root cause analysis and continuous improvements.

On this regard, it is important to realize that medical devices regulations do not mandate a specific format for records, so in principle maintaining records in a tool is not prohibited. It is the responsibility of the team to select tools that are fit for the purpose, to ensure that the integrity of the information is preserved, and to ensure that any applicable regulatory requirements are met, including consideration whether records are readily accessibility in accordance with the intended use or need.

When selecting tools, it is therefore of primary importance to take into account both aspects: teams needs and regulatory expectations. The best tools will satisfy both, not be viewed as favoring one to the detriment of the other.

The following benefits brought by tool automation are particularly relevant for teams implementing **AGILE** in a regulated environment:

— **Change management.** Software tools can be of great help in establishing a good balance between the **AGILE** culture of embracing frequent changes and the regulatory need of controlling and appropriately documenting them. A well-selected tool would provide a simple interface for recording all needed information related to the change and reduce possibilities of human errors by, for example, providing predefined options to choose from (e.g., drop-down menus, guided editors). In addition, a good tool would provide means to detect inconsistencies, track history of the changes, extract metrics, and detect trends. The team may review this information as part of their RETROSPECTIVES, to determine potential improvements and establish if corrective actions are needed.

— **BACKLOG management and TRACEABILITY**. **BACKLOG** management is essential for **AGILE** teams to manage the INCREMENTAL development model where work is performed in many small batches, especially for large and geographically distributed teams with very large **BACKLOGS**. With the support of tools, **BACKLOG** items are made accessible at any location and can be updated by any member of the team. Functionalities like virtual dashboards can be useful to automatically calculate and visualize important information such as velocity, burndown, control charts and other **AGILE** metrics used by the team. From a compliance perspective, using a tool that enables the creation and maintenance of TRACEABILITY between items (such as **BACKLOG** items, DESIGN INPUT/OUTPUT, defects, design changes, risks, mitigations, test cases, etc..), may be very valuable,

especially when it facilitates the detection of potential gaps (e.g., via alerts, queries, or dashboards) that may result in quality issues.

— **Documentation**. When operating in a regulated environment, **AGILE'S** goal of frequently delivering working software also implies the delivery of related documentation. Using tools that facilitate creation and modification of design documentation so that software development and documents proceed at the same pace, is critical to enable continuous delivery. Additional tool functionalities that may further facilitate documentation management include electronic approval and extraction of specific design information (e.g., via queries) to show on request, and the ability of linking different design controls items (e.g., design inputs, outputs, test cases, VALIDATION, etc.) that belong to the same software RELEASE.

Choose tools that are able to satisfy both the development team and regulatory **STAKEHOLDERS**.

5.7 Human safety risk management

Regulations require medical device companies to follow a robust set of human safety risk management activities in their product development. Such activities include risk planning, risk analysis, risk control identification, and risk control **VERIFICATION**. The documentation and approval activities should be in place in accordance with the organization's quality management system. The risk analysis is ongoing throughout the project so as to capture and deal effectively with any new risks that emerge.

AGILE does not provide practices related specifically to risk planning and risk assessment, but it does provide practices that integrate well with risk management activities. The risk analysis activities themselves can be managed through **AGILE'S BACKLOG** mechanism. As risk analysis is performed and risk control measures are identified (e.g., risk mitigations, labeling), they are input into the creation of **BACKLOG** items in the **STORY**, **INCREMENT**, and **RELEASE LAYERS**. The priorities of these **STORIES** are set to the appropriate level to ensure that they are dealt with before the project's completion.

Integrate risk analysis and risk mitigation into ***STORY*** *creation and* **BACKLOG** *management activities.*

These risk controls must be verified and any new risks that emerge as the design progresses must be captured and dealt with.

AGILE'S DONE is **DONE** concept details the requirements for completion of work in a **STORY**, **INCREMENT**, and **RELEASE LAYER**. In the **STORY LAYER** a team can ensure that a risk control is properly implemented and verified for that **STORY**.

Use **DONE** *is* **DONE** *criteria in the* ***STORY, INCREMENT,*** *and* ***RELEASE LAYERS*** *to ensure that risk controls are implemented and verified.*

Any new risk that is identified from the **STORY** can be added to the **BACKLOG** for proper disposition within the risk process. In an **INCREMENT LAYER**, as all **STORIES** are completed, risks from **STORY** interactions can be identified and added to the **BACKLOG**. And in a **RELEASE LAYER**, **DONE** is **DONE** ensures that all risk activities are completed and verified.

Reevaluate safety risk as ***STORIES, INCREMENTS,*** *and* ***RELEASES*** *are completed.*

There are some additional benefits that **AGILE** methods bring to ensure effective human safety risk management. For example:

— Team collaboration practices ensure focus on safe design.

— Frequent customer interaction/**VALIDATION** provides feedback on the usability and safety of the product.

— Continuous integration gives visibility to the effectiveness of the risk control measures.

6 Aligning on practices

This section addresses some of the detailed issues to be considered when applying **AGILE** to medical device software. This section is organized by these perspectives:

- Process requirements of IEC 62304, and how **AGILE** practices might address them;
- Selected **AGILE** practices, and the regulatory considerations for them;
- Process requirements from other regulations and standards, and how **AGILE** practices might address them.

6.1 Addressing IEC 62304 in an AGILE way

This section, organized by sections of IEC 62304, gives guidance on how **AGILE** practices should be considered when addressing the requirements of IEC 62304. See also Appendix B for a simplified view of applying Agile to IEC 62304.

6.1.1 Topics related to planning

6.1.1.1 Is AGILE too undisciplined to meet planning requirements?

AGILE approaches are sometimes perceived to lack disciplined processes and process execution. This perception is not correct.

AGILE has the concept of layers of planning, with planning occurring in the **STORY**, **INCREMENTS**, and **RELEASE LAYERS**. In the **STORY LAYER**, there is a plan for elaboration and implementation of a specific piece of functionality. In the **INCREMENT LAYER**, there is a plan for all of the functionality that will be added in a fixed time window. And in the **RELEASE LAYER**, there is a plan for all activities that are needed to finish the software development project.

AGILE has the concept of just-in-time planning. At the beginning of a project, **AGILE** teams will normally lay out a **RELEASE** plan (e.g., **BACKLOG** of features, number of **INCREMENTS** needed to implement these features, broad level plan for each **INCREMENT**). Specific details for each **INCREMENT** are left to planning at the start of the **INCREMENT**. **AGILE** still does planning, but it is DONE just as it is needed, so that it can accommodate changes with minimum effort.

AGILE has the concept of a **DONE** is **DONE** checklist, which is executed and verified, and which provides feedback on the successful completion of the planned activity.

When looking at these **AGILE** planning concepts, there is nothing inherent that would prevent them from being used in a medical product development effort.

Regulatory requirements dictate specific subjects that must be addressed in their plans. An **AGILE** team will find that they need more than a **BACKLOG** and **RELEASE** strategy to cover some of these planning topics. They now will have to write formal plans around such subjects as testing (at all levels), risk management, and software configuration management. A good way to remain **AGILE** is to document the high-level strategy/resources/schedules/milestones and use the **STORY** creation/**BACKLOG**/**INCREMENT**/**RELEASE** management to plan and execute detailed tasks. Together, they form the software development plan for a project.

*In software development plans, incorporate regulatory requirements for planning in **AGILE**'s planning practices*

6.1.1.2 Is shippable software, after every INCREMENT, a realistic expectation?

AGILE practices clearly drive the expectation that the team will deliver shippable software after each **INCREMENT**. For some medical device development efforts, this might be challenging and perhaps impractical. Some activities needed to make the software shippable, such as final system integration, formal **VERIFICATION** and **VALIDATION** (V&V), and the preparation of regulatory submission materials, might be difficult to complete each increment.

To address this challenge, **AGILE** teams should first consider what could be done to reduce the difficulty of completing all activities within an **INCREMENT**. Investment in infrastructure might help, such as test automation to make VERIFICATION testing and regression testing fast enough to complete every **ITERATION**. Changes to procedures might help, such as streamlining review procedures to encourage review/approval cycles to take hours instead of days. Changes to philosophies or assumptions might help, such as replacing the expectation of executing all tests during a "final formal V&V cycle" with INCREMENTAL test execution strategies.

*Improve processes, tools, and infrastructure to make it possible to deliver shippable software every **INCREMENT***

Even with such improvements, it might still be difficult or undesirable to produce shippable software every increment. In this case, Increment planning activities should determine which increments are intended to produce shippable software and which are not. To align with the **AGILE** concept of a DONE is DONE (or "Definition of DONE"), there might be a "Definition of DONE" checklist for Increments that are not intended to produce shippable software, and an additional "Definition of DONE" checklist for those that are.

***INCREMENT** planning should address which **INCREMENTS** should produce shippable software.*

6.1.1.3 AGILE'S focus on "working software" and continuous integration forms a very effective integration strategy

During planning, a team must decide on an integration strategy. **AGILE'S** focus on working software and continuous integration forms a very effective integration strategy. More specifically, **AGILE** strongly encourages the development of architecturally complete functioning slices of a system, to which further features, and functionality will be added during later **INCREMENTS**. Consequently, integration activities must be included in the software development plan almost from the ground up. These integration activities are captured in the **BACKLOG**. The software delivered at the end of an **INCREMENT** will typically include portions from all architectural layers, integrated and tested, rather than just a particular architectural layer, ready for another to be developed and integrated with it.

AGILE approaches require continuous integration and testing of all software components (including SOFTWARE OF UNKNOWN PROVENANCE [SOUP]), which largely manifests itself in team working practices (it's just how the team does their job), explicit integration, and the test activities that will be included in the **BACKLOG**.

*Use **AGILE'S** continuous integration practices as the core of an effective integration strategy.*

6.1.1.4 AGILE'S DONE is DONE concept is core to creating a VERIFICATION plan

Medical product development requires a **VERIFICATION** plan. How does **AGILE** facilitate this planning?

The readiness principle at the end of an **INCREMENT** or **RELEASE** requires that all development procedures be addressed in that timeline; thus, those procedures must be a part of the planning activity. In practice, this means that these activities take place at a low level in the daily work routine of the team.

To achieve this for each **STORY** the team must address the **STORY** details (requirement analysis), the architectural and detail design, the software unit implementation and **VERIFICATION**, the software integration and integration testing, and the system testing.

All deliverables are included in the **BACKLOG** and hence form part of the software development plan. Activities are often split into tasks, including **VERIFICATION** tasks that are needed for the activity to satisfy the **DONE** is **DONE** checklist. Every deliverable is subject to a **DONE** is **DONE** checklist, although different types of deliverables (e.g., software or user documentation) can have specialized checklists as appropriate.

INCREMENTS provide the most natural milestones in an **AGILE** project at which deliverables are verified. The **BACKLOG**, which contains deliverables including estimates of remaining work, allows the team to see at which milestone (**INCREMENT**) a deliverable will be completed. In the **AGILE** approach, a deliverable is not complete until it has been accepted.

This operates at two levels in **AGILE** projects:

1) A checklist for each significant class of deliverable (e.g., software or user documentation). This describes the generic acceptance criteria that apply to deliverables of that class; and

2) Specific instances of acceptance criteria will be agreed upon between the developer and the customer for each deliverable. These criteria might be captured as task breakdowns for the deliverable. This is usually performed during **INCREMENT** planning.

AGILE users have the option of following what can be called "the daily **BUILD** principle." The daily **BUILD** principle says that work items that are believed to be working can be "checked in" for the daily **BUILD**. This rule facilitates all the **VERIFICATION** activities that occur afterwards because it creates an environment of running automated tests at night and getting early feedback of any test results, which is an important part of the process for getting a working item at the end of an **INCREMENT**.

As a note, the practice of continuous integration typically requires that configuration management practices be defined and perfected very early in the project and then relentlessly followed. Failure to do so will be highlighted by the "daily **BUILD**" failing in a very public way. The requirement for configuration management practices to be in place from day one of a project means that the potential for errors or omissions is greatly reduced by comparison to more traditional approaches.

This also gives immediate feedback to the team when things go differently than planned. By having early test results, the team could detect at an early stage, if they forgot to address anything during the planning process, allowing the **INCREMENT BACKLOG** to be adjusted. In practical terms, this means that sometimes some items previously selected for an **INCREMENT** will be returned to the **BACKLOG** and the development plan is updated on the spot.

AGILE approaches, however, provide some key benefits by comparison to traditional approaches. Their iterative nature means that design and development **VERIFICATION** is generated continually, typically every 1-to-4 weeks, almost from day one of the project. Most of that **VERIFICATION** is based on working software and documentation, in contrast to traditional approaches in which often only documentation is available for review throughout much of the development.

VERIFICATION of tools or methods is just another **BACKLOG** item and can be planned for by placing in the **BACKLOG**.

*Use **AGILE'S** **DONE** is **DONE** concept as the core of an effective **VERIFICATION** plan.*

6.1.2 Topics related to software requirements analysis and documentation

6.1.2.1 Relationship between Stories and requirements

In medical device software development, requirements are a definition mechanism. Requirements describe what the product does and what attributes it has, and design verification activities are performed to confirm that the product meets its requirements. Requirements must be maintained to reflect the state of the product. As requirements are changed, configuration management is needed to control how different versions of requirements line up with different versions of the product and VERIFICATION activities for that product. (IEC 62304 has further requirements for the process of managing software requirements.)

In AGILE, Stories have a dual role as a planning mechanism and a definition mechanism. They are used by AGILE teams to define, plan and execute their work.

As a planning mechanism, Stories contain information such as when the STORY will be delivered (what **ITERATION**/**INCREMENT**/**RELEASE**), it's priority and relationships to other Stories, who will deliver it (what team or specific roles), an estimate of size (effort), and sometimes detailed information on the tasks to be performed in order to call the STORY DONE. As with other planning artifacts, once development is complete and the product is delivered, the planning content in the STORY becomes less useful, and therefore it is typically not maintained or updated as development proceeds. When considering Stories as only a planning mechanism, **AGILE** teams typically have a rule that says once a STORY is closed, it is not re-opened—once a STORY is completed by delivering something, if that something is later changed, a new STORY is created.

As a definition mechanism, Stories contain information that defines what is to be delivered and a description of the attributes or characteristics of the solution to be delivered. This definition information can be conveyed in different ways within a STORY. It could be in the description of the STORY (such as in its title or descriptive text), augmented by the conversation that takes place between the Product Owner and Development Team as the STORY is planned and delivered. It could be in the acceptance criteria of the STORY, which could be in the form of simple descriptive text to facilitate the conversation between the Product Owner and Development Team, or it could be in formal requirements language such as that described by requirements management standards. Finally, the definition information could be links to the formal requirements contained in a requirements document or requirements management tool, separate from but referenced by a STORY.

To provide guidance to development teams and avoid confusion within the teams and/or by external auditors, development plans and procedures should clearly define how Stories and requirements are used in defining the product to be delivered.

In plans and procedures, define how Stories and requirements are used in defining the product.

(While this section addresses the relationship between Stories and requirements - there is similar relationship between Stories and user needs. This is addressed in Clause 6.3.1.)

6.1.2.2 Stories and Requirements: One thing or two?

Do Stories and requirements have to be separate but related things, managed differently, with different tools, and having different structures? Or can requirements be embedded into Stories as a single thing managed by a single tool? The answer depends on a few factors for AGILE teams to consider in making their decision:

- Formality of the requirements documentation: AGILE encourages Stories (their definition and their acceptance criteria) to contain “just enough” definition to guide the team, so for organizations where that level of definition is sufficient, there may not be a need for anything else (as long as they satisfy expectations for requirements as defined in IEC 62304). If organizations find value in formally structured requirements such as those defined by requirements engineering standards, applying that level of formality might be better managed in a separate requirements repository.

- TRACEABILITY: As required by IEC 62304, software requirements must be traced to the higher level system requirements, risk control measures or other source, as appropriate, and must be traced to tests or other VERIFICATION methods. TRACEABILITY mechanisms are common in requirements management systems, but might not be as common in BACKLOG management systems, so if a STORY contains requirements, TRACEABILITY mechanisms must also be built into the STORY.

- Creating a complete requirements document: It is often necessary to generate a complete “document” of all requirements for the software product, such as for a final review and approval of all requirements for the product to be RELEASED, and for regulatory submissions. If all requirements are documented within Stories, the BACKLOG management system must have a mechanism for aggregating them all into a “document”.

- Maintenance and configuration management: As with other artifacts of software development that change as the product is developed, as requirements change it must be possible to show which versions of requirements go along with different versions of the software and associated tests. If requirements are documented within a STORY, this can be a challenge, especially when an AGILE team has a rule for not changing Stories once they have been closed. The degree of difficulty depends heavily on the backlog management tool. If Stories are managed with index cards, configuration management of the requirements content within a STORY is practically impossible. If Stories are in robust backlog management tools that can manage the integration with artifacts that need configuration management, maintenance of the requirements could be performed.

Combining all these factors, the decision on whether there are two separate artifacts for requirements and Stories, or whether requirements can be embedded into Stories depends on the organization’s expectations for the rigor of the requirements management process and the tools used to manage backlogs and requirements. If Stories are used as

requirements, the expectations for requirements management and the requirements of IEC 62304 for Software Requirements Analysis must be satisfied, while not unduly burdening the mechanism of AGILE STORIES.

*If **STORIES** are used as requirements, plans and procedures should define how the Software Requirements Analysis process of IEC 62304 will be satisfied.*

6.1.2.3 When does a STORY have enough definition to begin work?

A **STORY** is a high-level description of a piece of functionality of the system. It must be detailed enough to allow the team to prioritize and the developer to effectively elaborate and implement the STORY, so the team will have a key role in giving feedback on the DONEness of a STORY definition. Effective prioritization helps ensure that the highest value stories or features are performed early in the development effort. The presence of an embedded customer/product owner will facilitate the elaboration of the STORY with minimum documentation.

__STORIES__ must be detailed enough to allow the developer to effectively elaborate and implement the __STORY__.

6.1.2.4 Requirements DONE when a STORY is DONE

For all requirements in the scope of the **STORY**, it is expected to complete the requirements as part of the **STORY**. Complete in this context means documented, controlled, and reviewed/approved, acknowledging that complete does not mean the requirement is locked down from further change. There may be reasons for the requirements to not be complete when the **STORY** is finished. A team may choose to define **DONE** at a **STORY** level as documented and controlled but will do final review and approval at the **INCREMENT**/**RELEASE** level. In any event, the **DONE**ness criteria for the **STORY** should identify whether it needs to be documented, whether it needs to be controlled, and whether it needs to be reviewed/approved. When **STORY** work does not complete a requirement, such as deferring the approval, the **INCREMENT** or **RELEASE LAYER** work will need to complete that work.

Requirements will change more often in **AGILE** than in a linear model, so teams must ensure that they employ a change management process that can effectively handle frequent change.

*Use the **DONE**ness criteria for a __STORY__/__INCREMENT__/__RELEASE__ to identify where a requirement is formally documented, controlled, and reviewed/approved.*

6.1.2.5 Requirements documentation

There is an extensive list of requirements around the contents of a product requirements specification. All of these criteria should be addressed as part of the creation of the final product requirements. See the FDA guidance and IEC 62304 standards for details.

Existing requirements documentation techniques, such as use cases, textual descriptions, screen mock-ups, control flows, etc. are all compatible with **AGILE** and **AGILE** does not suggest a need for a specific technique.

Demonstrate that requirements documentation covers all areas that are required by regulations.

6.1.2.6 Can EXECUTABLE REQUIREMENTS be a valid part of the requirements definition and documentation?

EXECUTABLE REQUIREMENTS are an **AGILE** technique where requirements are written in a formal language such that the test for the requirement is automatically generated. This is a natural extension of **ATDD**, where the test is written before the code is developed. In this case, the test is automatically generated when the requirement is written. Following the concept of **ACCEPTANCE TEST-DRIVEN DEVELOPMENT**, code is added to the product until the test passes. The obvious advantage is the labor savings that comes from automated tracing/test development and the enhanced quality that comes from immediate feedback on their code implementation.

The challenge comes with providing a reviewable requirements documentation package to regulators. A team must ensure the readability of whatever requirements are pulled from the set of **EXECUTABLE REQUIREMENTS**. Use of formal words in your requirements definition language that are more English-like will facilitate this. Also use of tools to package and organize requirements in a comprehensible outline will help this.

EXECUTABLE REQUIREMENTS *can be used as part of a final requirements documentation mechanism.*

6.1.2.7 How does AGILE address requirements VERIFICATION and VALIDATION?

There are several ways in which **AGILE** methods and practices contribute to the process of requirements VERIFICATION and VALIDATION:

AGILE promotes the active involvement of the customer (or customer proxy) by making the role part of the project team. The customer role is involved in the writing of **STORIES** and acceptance tests. This helps with determining priority of **STORIES** and ensuring that the requirements are better understood by removing the risk of ambiguity and misinterpretation.

Often getting access directly to customers is difficult so a customer proxy (e.g., marketing representative) will represent the customer's view. In this case it is especially important for them to embrace the process and be prepared to commit to the time required to sustain frequent or nearly continuous contact with other members of the development team. **AGILE** teams are served well by identifying this need up-front in a project so as to ensure management support for this role.

The practice of pair programming helps with the process of requirements VERIFICATION by enabling pairs to refine the requirements as the **STORY** is implemented.

Demonstrations are a mechanism for feedback from the users. The customer can be involved in the demonstrations so as to validate the adequacy of the requirements. This technique could also help reduce the risk of incorrect requirements definition from using a customer proxy.

Use **AGILE** practices for continuous requirements ***VERIFICATION*** and ***VALIDATION***.

6.1.3 Topics related to software architecture

6.1.3.1 Evolving architecture

In **WATERFALL**-like models, architecture is an activity which usually begins early and ends just prior to other design activities, such as coding. **AGILE** methods recognize the reality of "**EMERGENT** architecture" and recommend avoiding a "big up-front design" activity that includes specifying and locking down an architecture too early. Architecture is an activity that permeates the entire life cycle of a product.

This **EMERGENT** architecture is easier on smaller systems. Larger systems might require more architecture up front and are less receptive to significant **EMERGENT** changes.

In practical terms, however, and since each development is unique, expert judgment is required to decide how much of the architecture must be defined at each point in the development: the architecture for the whole system, just enough for the next slice of the system to be delivered, or some balance.

The **AGILE** architecture is addressed in two layers: the **PRODUCT LAYER** and the **STORY LAYER**.

— In the **PRODUCT LAYER** big picture architecture activities are performed, enough to establish a framework to plan **RELEASES**, **INCREMENTS**, and some **STORIES**, and then to execute some **STORIES.**

— In the **STORY LAYER** specific impacts on the architecture are addressed. As activities are performed in the **STORY LAYER**, the architecture may guide the way the **STORY** is implemented, and the **STORY** may suggest changes needed to the architecture. If the architecture is to be changed, then the **STORY** should address how that change is to be documented and approved.

Initially, foundational architectural elements may be set in early **ITERATIONS**, with the architecture evolving in every **ITERATION**, as necessary.

A software architect may be needed to drive the architecture and provide input to the team when required. The role also takes the lead in the review of the architecture, however, from an **AGILE** perspective, architecture is a team effort. Every team member is responsible for architecture as it evolves to support the product features, create team-wide common knowledge, and identify architectural problems.

*Assess how much architecture work is needed in the context of a **STORY**.*

Thus, the team also needs expertise in evolving design distributed among the teams and requires good coordination between teams when interface definition crosses team boundaries.

6.1.3.2 Architecture planning

Software development plans should address:

- Where architecture work happens (part of a **STORY**, a **STORY** by itself, or work separate from a **STORY**).
- Who defines and documents the architecture, especially when multiple teams are working on a project (one team, all teams, a separate architecture-focused team)?
- When the architecture documentation gets produced (on a team with architecture expertise that is evolving an architecture, up-front documentation is less necessary–on a team where architecture work is done separately from the development team, up-front documentation is more important).

At regular points throughout the development, and certainly at **RELEASE** points, the requirements and architecture (for the part of the system being delivered) must be complete, consistent, correct, documented at an appropriate level and approved. A finish-to-finish relationship (where the requirements must be finished before the architecture can be finished) exists rather than the traditional finish-to-start relationship (where the requirements must be finished before the architecture definition can start).

In software development plans, specify when architecture work is performed.

6.1.3.3 Architecture **VERIFICATION**

The following **AGILE** practices can be used to get feedback on whether the architecture holds or has to be changed:

- **TDD** with continuous integration and automated testing;
- **REFACTORING**;
- Pair programming;
- **ITERATION** demos;

This rapid feedback on the successful implementation of requirements indirectly demonstrates that the architecture supports the requirements. When **REFACTORING** happens, the ease or difficulty of doing it is a reflection on the maturity and correctness of the architecture.

While pairing can be an element of architecture VERIFICATION because of the scope and risk involved, a pair is rarely sufficient; other review/VERIFICATION activities are necessary.

*Identify architecture problems early with **AGILE**'s emphasis on early and continuous **VERIFICATION**.*

Using these **AGILE** practices, it is guaranteed that the tested properties of the architecture remain stable while software is changed and extended. Additionally, it is recommended to use simulators (such as mocks or stubs) to isolate a dependency away from your system (e.g., the database, file system, user interface representation), or a real simulator for an instrument or other external device.

Continuous integration and VERIFICATION can provide valuable evidence that:

- **SOUP** is suitable for its intended use within the system; and
- The risks associated with integration and regression have been addressed.

When writing a design to mitigate previously identified risks, the information and the **TRACEABILITY** to the original identified hazardous situations should be part of the design document. It is important that future designers are able to understand the reasons for those choices and why the design is as it is. This is a way to avoid new changes that may leave the previously identified hazard without mitigation.

6.1.4 Topics related to detailed design

6.1.4.1 Activities of detailed design

Software detailed design is not necessarily a distinctly separable activity. In an **AGILE** model, the boundaries between the activities of architectural design, detailed design, and implementation are not as clear as linear-flow models might suggest. Recognizing that architecture influences detailed design which influences implementation as IEC 62304 describes, **AGILE** also recognizes that implementation can influence detailed design and that detailed design can influence architecture. While some up-front design work can be useful, **AGILE** rejects the concepts of "big design up front (BDUF)" or "big bang design" where a large design effort is completed before implementation begins.

In addition to design and implementation activities, other activities can also be thought of as design activities since they can influence the design of the software. Software requirements analysis, at the lowest level of **STORY** implementation, might make assumptions about detailed design. The structure and content of the software requirement documents can convey information about the software's design. VERIFICATION testing activities (at all three levels of unit, integration, or software system) also represent some level of design activity. **TDD** concepts encourage designs to be testable, with the tests themselves conveying information about the software's detailed design.

To show alignment with IEC 62304 and to satisfy regulatory expectations for performing detailed design, it is important for a software development plan or quality management system to define how detailed design is to be performed, even if the activity is not performed as a distinctly separable activity.

Define the activities that contribute to detailed design.

6.1.4.2 Emergent design

Software detailed design emerges, small bits at a time, as functionality is added to the software. **AGILE** emphasizes **EMERGENT** design. As new functionality is added to a software system, the design emerges to support the new functionality. Using the principle of "simple design," **AGILE** encourages designs that are simple enough to deliver the functionality of the current **STORY**, and discourages complex designs intended to provide options for future functionality that is not yet in the product.

Like the detailed design of software units, interfaces (external and internal) emerge as new functionality is added to the software system. Interface definition occurs sooner and more robustly in an **AGILE** model, since the breaking down of a system into small pieces requires the interfaces to be well-defined to make the small-piece model work. In a system with poorly defined architecture and interfaces, the development of small pieces could lead to chaos, a poor design, and excessive rework/**REFACTORING**.

Consistent with **AGILE'S INCREMENTAL/EVOLUTIONARY** life cycle, detailed design of software units and interfaces is most often performed in the **STORY LAYER**, with small bits of the detailed design being created as needed for the scope of the **STORY** being developed. Less often, some detailed design can be performed in the **PRODUCT LAYER** to explore design options and mitigate project risk.

Using the principle of "**REFACTORING**" as a companion to simple design, **AGILE** encourages designs to be reworked as new functionality is added and better designs emerge. Recognizing the difficulty and risk of trying to anticipate all future

needs in an up-front design, **AGILE** encourages design changes as the need for changes are identified and well understood to support new functionality.

Since **AGILE** work is organized by **STORIES** that represent delivered value and not necessarily organized by software units, the detailed design of any software unit may change as the addition of new functionality suggests that it should. The design for any software unit is "**DONE**" only to the extent that the functionality it supports is "**DONE**."

A concern with **EMERGENT** design, especially in **AGILE'S** highly iterative/**EVOLUTIONARY** life cycle, is that even if the pieces are designed well, they might not come together to produce a complete, consistent, and high-quality design for the software. To mitigate this concern, it is important to perform regular **VERIFICATION** of emerging design.

*Perform regular **VERIFICATION** of emerging design.*

6.1.4.3 Documentation of detailed design

The **AGILE** value statement that emphasizes "working software over comprehensive documentation" is particularly relevant to detailed design. While documentation of detailed design is important to various STAKEHOLDERS, more important is that the software is well designed. **AGILE** emphasizes the value of the design activity and encourages design teams to challenge the value of the documentation being produced. Well-written code using modern programming languages along with a complete set of well-written tests can itself be good documentation of the detailed design—separate documentation may not always be useful or necessary. Regardless of where the documentation exists, it should be a useful by-product of the design activity, useful first to the team itself, and then also usable by regulatory STAKEHOLDERS.

Regulators have different needs and expectations for design documentation than the design team itself—what is useful to the development team may not be useful or understandable by a regulator. This may mean that well written code alone may not be sufficient to demonstrate detailed design to a regulator. When defining the detailed design documentation to be produced, consider the needs of different users of that documentation.

Define the detailed design documentation to be produced, considering the needs of development teams and regulators.

For more information about documentation, see Clause 5.5 Documentation.

6.1.5 Topics related to implementation and unit VERIFICATION

Planning of implementation activities when developing software using **AGILE** methods begins with planning activities in the **PRODUCT LAYER, RELEASE LAYER**, and **INCREMENT LAYER**. Implementation activities begin when the project team selects a **STORY** to be delivered in an **INCREMENT**. Some **AGILE** practices related to implementation and unit VERIFICATION which are described in this clause includes:

- **TDD**;
- Pair programming;
- Continuous integration;
- Continuous automated testing.

All of these practices foster collaboration among **AGILE** project team members and a disciplined development process. (See more on **TDD** in Clause 6.2.11)

TDD is an implementation method whereby the developer creates the unit test to ensure the code performs as desired, prior to writing the code that implements the functionality for the **STORY**. This ensures that every bit of code that is written is tested at a design level and that the tests are repeatable (usually automated) resulting in higher quality code.

Pair programming is an implementation method whereby two developers sit side-by-side and write code. Doing so increases the code quality, understanding of the task at hand, and ultimately resulting in less rework. This activity is

focused on finding coding defects, coding inefficiencies, lack of adherence to standards, including coding guidelines, and enhancing the general understanding of the part of the **STORY** being implemented. In addition, it is a very disciplined way of developing software as two people must approve the code being implemented prior to it being integrated into the main code base. (See more on Pairing in Clause 6.2.10)

Implementation and unit VERIFICATION activities are performed on software units, in an environment of continuous integration and integration testing. The practice of continuous integration results in frequent **BUILD**s of the code base. The goal is simple, to detect anomalies early, so the code base is stabilized very early in the project. Contrast this with the method of implementing many units before integrating and testing them. This method is prone to a higher number of conflicting changes by developers, a larger number of defects, and a larger amount of change in the software code base rendering it unstable for a longer period of time while all issues are fixed and retested. Frequent and continuous integration and testing demonstrates that the software is built to be stable and maintain its stability as new functionality is added. (See more on Continuous Integration in Clause 6.2.9)

From an **AGILE** perspective, being able to demonstrate working software, a **STORY** at a time, is critical to developing and finalizing the detailed software requirements and other artifacts for each **STORY**.

Use **AGILE** *practices such as* ***TDD****, pair programming, continuous integration, and continuous testing to deliver high quality software.*

These **AGILE** practices are early quality indicators and may result in the team declaring “stop the line” if early test results or integration indicate there is a quality issue. The project team must fix the failure(s) before additional code can be integrated. This results in a controlled and stable code base.

6.1.6 Topics related to integration and integration testing

The process of integration is when various software units or even software applications come together to form a system. In many cases, integration will also include hardware components, especially if the software module or product is firmware, which is part of a hardware product. Integration testing is the process of verifying that all these software and hardware components function as required.

When using **AGILE** practices and methods, integration and integration testing are activities that are performed continuously and usually begin very early on. Right from the outset, when **STORIES** are being written, the **AGILE** project team will use **TDD** techniques to write detailed design tests and perhaps even **EXECUTABLE REQUIREMENTS**. They will then setup a continuous integration process which will, over time, execute these tests automatically each time changes are made to the product at code level. The process can be setup to incorporate all software units or applications that make up the system. Depending on the complexity of the system, it may be difficult to write unit tests or **EXECUTABLE REQUIREMENTS**. For example, the system may contain many hardware components that are required to perform certain tests. Today, there are several techniques used by **AGILE** project teams such as, creating hardware emulators or stubs that make the inclusion of hardware components possible. It is important that project teams evaluate the system complexity and determine to what extent the continuous integration process covers the integration and integration testing of the system in question.

It is far more common that the continuous integration process involves the **VERIFICATION** of the integration of software units and that this is performed in the **STORY LAYER**. **VERIFICATION** of the integration of complex systems and/or broader system components is more likely to be performed in the **RELEASE** or **INCREMENT LAYERS** in which a certain degree of manual integration testing may be required. Regardless of the integration process used, it is important to ensure that the integration and integration processes be undertaken regularly and that they occur in the **STORY**, **INCREMENT** and **RELEASE LAYER**s. Project teams can also utilize the continuous integration practice to perform early and regular integration of the units or components that present high technical risk or risk related to harm. Within all three layers, the content and output of integration testing can also be input to the **DONE**ness criteria of the **STORY**, **INCREMENT** or **RELEASE** for the product or project.

Use **AGILE** *practices of* ***TDD*** *and continuous integration to perform integration and integration testing early and often.*

While the practices of **TDD** and continuous integration are very useful for regression, it is not sufficient since errors in the integration process are often not in things that were specified in a requirement or an automated test. Therefore, there is still value in performing manual testing to identify errors in integration.

(See more on Continuous Integration in Clause 6.2.9)

6.1.7 Topics related to software system testing

Software system testing is the activity that verifies the software's functionality and performance with respect to its requirements. Regulations and standards require that this type of testing be performed on the final software product, but they also suggest that software **VERIFICATION** and **VALIDATION** be integrated into the entire software development cycle. **AGILE** reconciles these expectations by addressing software system testing in three layers of abstraction, the **STORY**, the **INCREMENT**, and the **RELEASE LAYERS**.

In the **STORY LAYER**, software system testing is focused on verifying the functionality covered by the **STORY**, with tests that demonstrate the requirements related to that **STORY** have been met. In the **INCREMENT LAYER**, multiple **STORIES** come together to provide a set of related functionality, where software system testing is focused on the interactions and interfaces of the smaller pieces of functionality, which may be defined by requirements that span more than one **STORY**. In the **RELEASE LAYER**, where the entire software product comes together, software system testing is focused on demonstrating that the entire product is meeting all requirements.

6.1.7.1 The importance of software system test planning

Given the differences between an **AGILE** testing model and a linear model, it is important for software development plans to clearly describe the different layers of testing. Furthermore, because so much testing is performed on different versions of the software, it is important for software development plans to clearly describe what testing is performed on which software versions, and how final testing demonstrates that final version has been thoroughly tested.

Describe the different layers of Software System Testing in software development plans.

6.1.7.2 The value of continuous testing

An advantage of the **AGILE** model where software system testing occurs as code is developed is that there is continuous feedback on the correctness of the software, so errors in definition, design, or coding can be exposed quickly, and therefore addressed more effectively. In a linear model, where software system testing occurs after the software product is complete, errors are exposed later in the development cycle where they are harder and riskier to correct.

Use continuous Software System Testing to get better feedback on the correctness of the software.

6.1.7.3 The importance of regression testing

A disadvantage of the **AGILE** model is that software system testing performed within the **STORY** and **INCREMENT LAYERS** are not necessarily repeated on the final version of the software product. In contrast, with a linear development model, it is easier to demonstrate complete coverage of testing when running all tests on a single version of the software. The concern with the **AGILE** model is that software might have been tested and shown to be correct in an earlier version, but that some later change introduces an error to code that was thought to be correct.

Regression testing addresses this concern. As with any **INCREMENTAL** model, regression testing is necessary to demonstrate that changes do not introduce unintended consequences in software that has already been tested. **AGILE'S** highly **INCREMENTAL**/**EVOLUTIONARY** model makes this even more important.

TDD encourages a high degree of test automation to address the need for regression testing. Tests run initially to demonstrate that new code is correct can also be run later to demonstrate the code stays correct. While automation works well, tests can be run as often as the test system can support it, giving continuous feedback on whether the software stays stable throughout changes.

Use Regression testing to ensure that errors are not introduced by **INCREMENTAL** *changes.*

Recognizing that fully automated and complete regression testing is often difficult to achieve, **AGILE** encourages thorough regression testing at **INCREMENT** and **RELEASE** boundaries even when that testing is not automated.

6.1.7.4 Tests are as important as the code

AGILE'S concepts of **ACCEPTANCE TEST-DRIVEN DEVELOPMENT (ATDD)** and **TDD** encourage teams to develop detailed requirements, tests, and code at the same time. The test first concept says that tests should be created before the code is created, as part of the detailed definition and design of the software solution. Regardless of the exact order of these activities, part of the completion of a **STORY** is the creation of the tests that demonstrate the code satisfies the requirements, thus enabling a means to measure what is intended. As part of **STORY** planning, teams should identify the tests that will be created and establish when the tests should be executed. The ideal case is to execute the test as soon as the code is written, but in some cases, it may be necessary to execute the test when some other part of the software is created in future **STORY** work.

Create software system test procedures when the code they are testing is created.

If code is changed in some future **STORY** work, the tests associated with it should also change through that **STORY** work. Tests are as important as the code to achieve high-quality software in a highly **INCREMENTAL/EVOLUTIONARY** life cycle.

6.1.7.5 Documentation of software system testing results

Regulations and standards require the results of software system testing to be documented. In an **AGILE** model where this testing is performed frequently as part of completing every **STORY** and performed continuously as part of a continuous flow of regression testing, the documentation work can become (or can seem to become) a burden that slows the team down. It is important for the documentation effort to be simple enough to be performed efficiently as tests are run and re-run. If a team is tempted to skip the documentation work, they should see that as a sign that the documentation activity is either not providing enough value or is too costly. As a result, the team should find ways to improve on both. There are several off-the-shelf tools that support automated reporting and trending of test results and once validated will satisfy the regulatory and standards requirements.

Create software system test documentation as a simple by-product of a robust test system.

6.1.7.6 TRACEABILITY to Software System Testing

Regulations and standards require that requirements be linked to VERIFICATION to show that every requirement has been verified. When the VERIFICATION is a test, **TRACEABILITY** needs to show that all tests have been executed. In the **AGILE** model, the links between individual requirements and their VERIFICATION are established as those requirements and tests are created as part of **STORY** work. The link between a test execution and the version of software on which it was executed is made when the tests are executed, either as part of **STORY-**LAYER testing, **INCREMENT-LAYER** testing, or **RELEASE-LAYER** testing. In addition, **TRACEABILITY** reports run at RELEASE boundaries demonstrate that all requirements link to tests that were executed.

The value in using **TDD** practices is that requirements, VERIFICATION, and test execution are very closely linked, thus essentially providing built-in **TRACEABILITY**.

*Integrate **TRACEABILITY** activities into the work* DONE *in the **STORY**, **INCREMENT**, and **RELEASE LAYERS***

If **STORIES** contain elements that need to be traced (such as requirements or tests), then **TRACEABILITY** to **STORIES** should be considered.

6.1.8 Topics related to software RELEASE

AGILE software development promotes the practice of short **RELEASE** cycles in order to allow project teams to get real end-user feedback on a regular basis, thereby driving down risks associated with not delivering on customer expectations. In the section 5.1.3 "Executing process activities in multiple layers of abstraction," the various process activity layers for a typical **AGILE** software development process were introduced. The **RELEASE LAYER** was described as

consisting of a series of activities that lead to the creation of a usable product. The output from the **RELEASE LAYER** could be a product that is intended to be shipped to the field or it may be a product that is meant for internal use only.

A key tenet of the **AGILE** practice of short **RELEASE** cycles is building software that is in a potentially shippable state. When placing this practice within the context of medical device software, it is often not practical to frequently ship a new software product when having to address regulatory approval requirements. However, for medical device software, short **RELEASE** cycles can produce a near-shippable product, in other words a product that is complete in every sense except the process of regulatory approval. One of the benefits of a near-shippable product is that it can be used for activities such as usability testing or controlled clinical evaluation ensuring that the product has real-world experience long before it officially ships after regulatory approval.

Clause 5.8.1 of IEC 62304 stipulates that before software is **RELEASE**d, all **VERIFICATION** needs to be completed and the results evaluated. **AGILE** practices make this step easier and a less surprising activity as each deliverable, activity or item is verified and documented as being verified within a single **INCREMENT**. This reduces the chance of significant issues appearing in **RELEASE**d product that can sometimes be the result of **VERIFICATION** activities executed in haste, due to project schedule pressures.

Before software **RELEASE**, all documentation must be complete. In the **STORY LAYER**, most of the documentation is created. In the **RELEASE LAYER**, the documentation is finished and reviewed to ensure that the sum of parts is an integrated whole.

Regulation also stipulates that before a **RELEASE**, all known residual anomalies need to be documented and evaluated. Although **AGILE** practices do not make special considerations for this requirement, short **RELEASE** cycles, and having near-shippable product at the end of each cycle, makes meeting this requirement less tedious and arguably more meaningful. **AGILE** practices, such as stop-the-line, support the process of having anomalies identified early and frequently and having them evaluated and fed back into the software development plan. Any anomaly that contributes to an unacceptable risk would normally be prioritized for fixing ahead of other features, as soon as it was identified. This lowers the chance of a substantial risk being identified at the end of a project when time pressures may result in compromises being made. **AGILE** teams can also use a risk-based approach to addressing bugs, deciding whether to fix them sooner to remove the risk, or whether they could be deferred with acceptance of the residual risk.

The frequency of building and releasing the software throughout the development life cycle of a product by means of **AGILE**-based automated process means that other considerations such as **RELEASE** documentation are made easier.

*Use **AGILE** practices to make the process of software **RELEASE** less risk prone and more efficient.*

6.1.9 Topics related to configuration management and change management

6.1.9.1 Software configuration identification

Although the IEC 62304 standard does not use the word “baseline,” it does require a team to document the set of configuration items and versions that comprise the software system. It would be prudent to baseline the software configuration at each major milestone or end of major phase, certainly at each **INCREMENT**, in **AGILE**. Baselining work products collectively is the key to understanding the maturity of a software system and to controlling changes. Changes to a baseline should consider the impact to the whole set of configuration items that make up the baseline. With continuous integration, this baselining would occur frequently and would consist of a more complete set of **ITERATION** life cycle deliverables. You might expect more deltas (changes to work products) than in a traditional project.

*Establish software configuration baselines at **INCREMENT** boundaries.*

6.1.9.2 AGILE’S impact on change control

In contrast to traditional software development approaches, **AGILE** facilitates change requests. A change request is logged in the **BACKLOG** and prioritized just as new features. Once selected for implementation, design control procedures apply the same for changes and new feature **BACKLOG** items during an **ITERATION**. Therefore, the

development process outlined by the IEC 62304 standard applies in the same way to change requests as to new features.

Regulations such as FDA QSR require manufacturers to identify, document, validate or where appropriate verify, review, and approve design changes before their implementation.

Satisfying this expectation may seem like a big challenge for **AGILE** teams, due to the amount and frequency of changes introduced during the development life cycle.

In fact, **AGILE** principles describe changes as something to welcome, to use for customer's competitive advantage and values more the ability of responding to changes over following a plan.

In contrast to traditional software development approaches, where change requests (especially at late stages) are perceived as something to avoid, when possible. **AGILE** teams embrace changes and are prepared to handle them throughout the whole development life cycle.

It is therefore important for **AGILE** teams to establish a LEAN change management approach that enables them to handle numerous and frequent changes in compliance with regulatory requirements, without creating a process bottleneck that could disrupt their agility.

In **AGILE**, a change request simply represents "work to be DONE" and therefore is logged in the **BACKLOG** and prioritized just as new features. Once selected for implementation, design control procedures apply the same for changes and new feature **BACKLOG** items during an **ITERATION**. Therefore, the development process outlined by the IEC 62304 standard applies in the same way to change requests as to new features.

Regression analysis is a requirement in IEC 62304. During implementation, a change must be evaluated for impact, and activities must be identified and repeated even to the extent of re-classification of the software safety class.

It is also well established that late changes to a software product bring inherent risks of introducing new errors. The change management system must be effective in evaluating the impact of changes on the system, and there must be robust regression testing to detect unintended impacts.

Use robust regression testing to detect unintended impacts.

Software tools may be of great help to ensure all needed information is recorded for change requests (such as impact assessments, regression analysis and needed communication activities).

A common dilemma in developing software in a regulated environment is to determine when to put the work products under change control. Critical thinking is required to determine the answer for each product, project, work environment, and circumstances. As a rule of thumb, when a work product is considered **DONE** (according to its "Definition of DONE") it should be placed under change control. Any further change affecting a **DONE** work product is considered a change request and logged in the **BACKLOG**.

As part of the "Definition of DONE*," determine when to put work products under change control.*

6.1.10 Objective evidence for activities defined in IEC 62304

In order to demonstrate compliance to regulations and standards, objective evidence must be produced. The objective evidence can be demonstrated as a "document" in a document control system or other artifacts managed in tools that support **AGILE'S** INCREMENTAL/EVOLUTIONARY life cycle.

In **AGILE'S** incremental/evolutionary life cycle model, documentation of objective evidence is produced throughout the life cycle, as described in section 5.5.3. Incremental evidence is created as part of completing a STORY, and aggregate evidence is created as part of completing an INCREMENT or RELEASE.

Development plans and/or procedures should specify the form and repository for objective evidence that is generated.

In plans and procedures, define the objective evidence to be produced for each software development activity.

The following table provides some examples of objective evidence to be produced for each of the software development activities defined in Section 5 of IEC 62304 when using an **AGILE** model (not intended to be an all-inclusive list).

Table 1—Examples of objective evidence

	BACKLOG Item/STORY	**INCREMENT**	**RELEASE**
Software Development Planning	Change records in a change management/**BACKLOG** management tool	Change records in a change management/**BACKLOG** management tool INCREMENT planning document Summary report pointing to relevant change management records	Software Development Plan RELEASE planning document Summary document pointing to relevant change management records
Software Requirements	Content in a requirements management tool, including trace links to/from requirements	Aggregate requirements document generated from content in a requirements management tool, including trace links to/from requirements Summary report pointing to relevant content in a requirements management tool	Aggregate requirements document generated from content in a requirements management tool Trace Matrix
Software Architecture	Content in a design modeling tool Sections of an architecture document	Aggregate architecture document generated from content in a design modeling tool Complete architecture document Summary report pointing to relevant content in a design modeling tool	Aggregate document generated from content in a design modeling tool Complete architecture document
Software Detailed Design	Content in a design modeling tool Sections of a document Code headers/comments Structure and contents of a source code management tool	Complete detailed design document Summary report pointing to relevant content in a design modeling tool and/or code Summary report describing the high-level structure of a source code management tool	Complete detailed design document Summary report pointing to relevant content in a design modeling tool and/or code Summary report describing the high-level structure of a source code management tool
Software Implementation (Code)	Content in the source code management tool	(N/A, addressed in Software Integration)	(N/A, addressed in Software Integration)
Software Unit VERIFICATION	Content in a unit test management tool Report from a static or dynamic analysis tool Code review indication in a source code management tool	Unit VERIFICATION summary report pointing to relevant content in a unit test management tool, static or dynamic analysis tool, and/or source code management tool	Unit VERIFICATION summary report pointing to relevant content in a unit test management tool, static or dynamic analysis tool, and/or source code management tool Unit VERIFICATION summary section of a broader VERIFICATION summary report
Software Integration	Software BUILD identifier (BUILD tag)/merge history in the source code management tool	Software BUILD identifier (BUILD tag)/merge history in the source code management tool Pull requests in the source code management tool	Software BUILD identifier (BUILD tag)/merge history in the source code management tool Software integration report (BUILD report)

	BACKLOG Item/STORY	INCREMENT	RELEASE
		Software integration report (BUILD report)	
Software Integration Tests (Procedures/Protocols & Results)	Content in an integration test management tool	Integration test summary report pointing to relevant content in an integration test management tool	Integration test summary report pointing to relevant content in an integration test management tool Integration test summary section of a broader VERIFICATION summary report
Software System Tests (Procedures/Protocols and Results)	Content in a software system test management tool, including trace links to/from Software System Tests	Software System test summary report pointing to relevant content in a software system test management tool, including trace links to/from Software System Tests	Software System test summary report pointing to relevant content in a software system test management tool Software system test summary section of a broader VERIFICATION summary report Trace Matrix
Approval (for any of the activities above) (See Clause 5.5.5 Approvals and evidence for more information)	Approval indication in a change management tool Approval indication within the tools that manage the content	Approval indication in a change management tool Approval indication within the tools that manage the content Signature on a document	Signature on a document Summary report (with signature) pointing to approvals in the tools that manage content

6.2 Using AGILE practices for regulatory compliance

This section examines some of the practices of common **AGILE** frameworks, addressing the relevance they have to regulators, and giving guidance on how to apply those practices in ways that align with regulatory expectations.

For all of the following practices, and any other **AGILE** practices that might be used, consideration should be given to how the practices are to be used, with emphasis on whether they contribute to the quality of the product and/or are relevant to activities required by regulations, since these factors determine their relevance to regulators.

For **AGILE** practices that have no direct impact on product quality (such as the team's rules for how a daily standup meeting is performed, or the technique used in a team RETROSPECTIVE), regulators are less likely to be interested in their details. These practices would not need to be described in procedures or plans.

For **AGILE** practices that have a direct impact on product quality (such as practices related to product definition, design, VERIFICATION, and VALIDATION), they are also likely to be related to regulations and therefore have regulatory compliance considerations. To demonstrate compliance and to define the practices in a way to ensure they add quality to the product, these **AGILE** practices should be defined in a procedure and/or development plan. This should not mean that **AGILE** practices must be defined as precisely followed recipes – they should be defined in a way that gives the development teams guidance on how the practice is to be used, while allowing the practice to be as flexible as it needs to be to ensure the practice is used effectively.

When **AGILE** *practices are used in an activity related to the quality of the product and/or regulatory compliance, those practices should be defined in procedures and/or plans.*

In addition to defining the practices, the objective evidence to be provided should be defined in procedures and/or development plans. Objective evidence is necessary to allow **AGILE** practices to be evaluated for compliance, such as during a regulatory compliance audit.

There are many ways for providing objective evidence, either direct evidence that demonstrates a practice was performed, or indirect evidence that infers the practice was performed. It could be:

A specific report, or part of a larger report, placed in a document management system.

The presence of other artefacts, such as requirements in a requirements database, code in a code repository, or tests in a test management system.

A review signoff that asserts a practice has been followed.

In any case, the objective evidence should be valuable to both the internal development teams and external auditors.

When defining **AGILE** *practices in procedures or plan, the objective evidence to be provided should also be defined.*

6.2.1 Product BACKLOG, BACKLOG Refinement, and "Definition of Ready"

The Product **BACKLOG** is the collection of all **STORIES**, the value-driven statements of features/functionality to be delivered. The Product **BACKLOG** can have as many **STORIES**, looking as far into the future as the Product Owner and **AGILE** Team are willing to elaborate. It is progressively elaborated, meaning the near-term **STORIES** are defined more clearly and completely while longer-term Stories could be more vaguely defined. The practice of **BACKLOG** Refinement is the ongoing evaluation of the Product **BACKLOG** to determine whether and when **STORIES** should be more completely defined. For a **STORY** to move into planning and execution, the **AGILE** Team establishes the "Definition of Ready," which is the agreement that indicates the **STORY** is sufficiently understood for the team to begin detailed planning and execution.

The **AGILE** practices of Product **BACKLOG**, **BACKLOG** Refinement, and "Definition of Ready" are related to defining the product, and are therefore relevant to the Design Input requirements of 21CFR820.30(c) Design Input and 13485 7.2.1 Determination of Requirements Related to Product and 7.3.3 Design and Development Inputs. As part of creating and refining the Product **BACKLOG** in preparation for planning and execution by the development teams, Design Inputs are being created and refined.

As described in Clause 5.3 Design Inputs and Design Outputs, in **AGILE'S** incremental/evolutionary life cycle model the relationship between Design Inputs and Design Outputs is more complex than a simple linear model of creating Design Inputs followed by creating Design Outputs. As **STORIES** are created for the Product **BACKLOG**, Design Inputs are considered. The level of consideration and the documentation of the Design Inputs varies depending on the context of the particular **STORY**, with options such as:

— When Design Inputs are likely to change or when they need elaboration as the design is created, the Design Inputs may be reflected only in notes within the **STORY** as reminders of the conversations that happened between the **AGILE** Team and the STAKEHOLDERS bringing the Design Inputs to the team.

— When there is value in providing explicit guidance to the development team, the Design Inputs might be documented as requirements or specifications in a requirements management system.

— When there is need to ensure agreement on the Design Inputs before embarking on product design, the Design Inputs might be reviewed and approved in a requirements management system and/or document management system.

The **AGILE** Team determines which of these is sufficient to meet the "Definition of Ready," deciding that the Design Inputs have been sufficiently understood for the team to begin detailed planning and execution of the **STORY**.

The "Definition of Ready" should be established in procedures or plans. The evidence that the **AGILE** Team has created the **BACKLOG**, refined the **BACKLOG**, and agreed that the "Definition of Ready" has been met should also be defined in procedures or plans, which could be the existence of Product **BACKLOG** items (**STORIES**) in the **BACKLOG** management tool.

"Definition of Ready" establishes that the Design Inputs are defined sufficiently to begin planning and execution.

6.2.2 Sprint BACKLOG

The Sprint **BACKLOG** is a planning mechanism of Scrum that captures the detailed planning and execution of a **STORY**. Adding to the value-statements of the **STORIES** to be delivered within an **ITERATION**, the Sprint **BACKLOG** also contains detailed tasks to be executed.

A Sprint **BACKLOG** is a mechanism for low-level planning, addressing some of the elements of IEC 62304 Clause 5.1 Software Development Planning. The detailed tasks could address activities required by the organization's design control procedures, such as low-level review-and-approval and testing activities, and other elements of the "Definition of DONE" (described below).

When using the Sprint **BACKLOG** as evidence of compliance to design controls procedures, a **BACKLOG** management tool is essential. (For more information about tools, see Clause 5.6.4 The role of software tools in regulated **AGILE**). For large projects that might produce hundreds or thousands of **BACKLOG** items (**STORIES**), each of which might have from a few to dozens of related tasks, the volume of evidence might be overwhelming for a regulator to analyze. When this is the case, a robust **BACKLOG** management tool that provides summary reports (with useful references to details) can demonstrate the organization's control over the design activities.

The Sprint **BACKLOG** *can provide objective evidence of compliance to design control procedures. Summary reports can* ITERATIONS and INCREMENTS

ITERATIONS are a time-boxing mechanism (such as the Sprints of Scrum), where small pieces of value are delivered through the completion of **STORIES** within a fixed time period (typically one-to-four weeks). **INCREMENTS** could be either another larger time-box (such as one or more **ITERATIONS**), or an **INCREMENT** of system functionality (such as the completion of a large piece of an integrated system).

ITERATIONS and **INCREMENTS** are mechanisms of low-level planning. They provide an opportunity to provide planning and reporting evidence that would demonstrate to a regulator that **AGILE** teams are in control of the fast-paced incremental-evolutionary model of **AGILE** development. Because this development model is not as simple as traditional linear life cycle models, regulators who may not be as familiar with incremental-evolutionary models might have concerns about whether the development team is in control of its design process. The periodic and frequent cycles of planning, execution, and adjustment of the **ITERATIONS** and **INCREMENTS** planning mechanisms provide the control that regulators require.

When defining the objective evidence to be provided, consider that it could be the organizational structure of the **BACKLOG** management tool or specific planning records in the **BACKLOG** management tool.

ITERATIONS *and* **INCREMENTS** *are planning elements that demonstrate to regulators that* **AGILE** *teams are in control of a complex incremental-evolutionary development life cycle.*

6.2.3 "Definition of DONE"

The **AGILE** concept of "**DONE** is **DONE**" means that when a **STORY** is delivered, it is expected that no more work is needed to complete the functionality of that **STORY**. "**DONE** is **DONE**" means the STORY has been defined (requirements and specifications written), implemented (design & code written), verified (reviewed, tested), and shown to comply with the design control procedures (documented, approved). To more clearly specify what is required for all of these elements, the **AGILE** Team creates their "Definition of DONE" ("DoD"), essentially a checklist of items to consider in planning and execution to determine which elements of DoD apply, and then to identify exactly how those elements will be satisfied. The relevant DoD elements often lead to specific tasks for the activities to be performed and documentation to be generated to satisfy the DoD.

DoD first applies to:

— **STORIES**, where small pieces of functionality are delivered, and where most of the detailed work of developing the software occurs. DoD elements for a **STORY** include such things as having specification and code artifacts checked into the configuration management system and reviewed and approved, unit-level tests executed, and pieces of relevant design documentation.

— Product increments, where larger pieces of functionality are delivered, such as through the integration of separately developed pieces of code into the larger software system. DoD elements for a product INCREMENT are things like having integration performed and documented and software system tests executed.

— Product RELEASES, where final activities are performed, and documents are finalized. DoD elements for a product RELEASE are things like having final documents approved in the document management system and summary reports written.

The "Definition of DONE" should be established in procedures or plans. The evidence that the **AGILE** Team has met the DoD for a STORY is, at a minimum, the closed Stories in the **BACKLOG** management tool, indicating that the **AGILE** Team has properly considered and applied the DoD. It might also include closed tasks from the **ITERATION BACKLOG**. Evidence for DoD for a product INCREMENT and product RELEASE might be a plan and report for the INCREMENT/RELEASE, or documents in the document management system. Additional evidence for specific DoD elements, such as owner and reviewer signoffs and document approvals, might also be generated, as defined by an organization's procedures and/or development plans.

"Definition of DONE*" establishes expectations for the activities and objective evidence for completing a* ***STORY****, Product* INCREMENT*, and Product RELEASE.*

6.2.4 ITERATION review

The **ITERATION** Review (an element of Scrum) is an opportunity for stakeholders to evaluate the team's completion of activities and deliverables from the current **ITERATION** and provide feedback to help plan and execute the next **ITERATION**. There are two parts to an **ITERATION** Review: planning and demo.

The planning portion the **ITERATION** Review provides an opportunity for the **AGILE** team to show the progress that was made in the completed **ITERATION**, to show how plans are proceeding, identify issues of interest to STAKEHOLDERs, and get input for the planning of the upcoming **ITERATIONS**.

While the **ITERATION** Review might be mostly a business planning practice that informs STAKEHOLDERS about the progress the teams are making (which is less relevant to regulators), it might include an evaluation of the design process that is more relevant to regulators. During the **ITERATION** Review, **AGILE** teams might discuss the effectiveness of their design control procedures to get feedback and help from STAKEHOLDERS.

The **ITERATION** *Review allows STAKEHOLDERs to evaluate the* ***AGILE*** *team's planning and execution mechanisms to ensure they are in control of their design process.*

The demo portion of the **ITERATION** Review provides an opportunity for the **AGILE** team to show the completed pieces of the product being developed, to get feedback on the product and adjust plans for future changes.

At the demo, the **AGILE** team asks the STAKEHOLDERS, "Does this product meet your need, does it perform as you expected, is it satisfying your intended use?" These questions are all elements of Design Validation. Though the demo does not eliminate the need for Design Validation activities that happen near the end of development before the product is RELEASED, it does provide important early feedback on the product to reduce the risk of finding major issues late in the development cycle.

The **ITERATION** *Review allows STAKEHOLDERs to provide early and frequent feedback on the product the* **AGILE** *team is developing – an important element of Design Validation.*

6.2.5 Product owner role

The Product Owner Role (an element of Scrum) is responsible for the **BACKLOG**, which includes the responsibility to gather User Needs from STAKEHOLDERS, refine those User Needs into software requirements, establish acceptance criteria for **STORIES**, prioritize and plan when **STORIES** will be delivered, and determine whether the delivered **STORIES** have met the acceptance criteria. The person in the role does not necessarily do all of those activities alone, they usually get help from other development team members and STAKEHOLDERS, as necessary. The Product Owner Role is

the primary interface between the STAKEHOLDERs who want the product and the development team that designs and delivers the product.

There are two elements of Design Controls that are particularly important to the Product Owner Role: Design Inputs and Design Validation.

In contrast to linear models where Design Inputs are defined up front and then implemented by the development team, within **AGILE'S INCREMENTAL/EVOLUTIONARY** life cycle model, Design Inputs are identified and refined throughout the development effort. The Product Owner Role is responsible for managing this. At the beginning of a development effort, the Product Owner Role ensures User Needs are identified and defined sufficiently for the development team to begin development of new features and functionality. Additional Design Inputs may be identified through ongoing interactions with STAKEHOLDERs to identify new and changing User Needs or by gathering new insights into the product as its design emerges. The Product Owner Role ensures that new and changed Design Inputs are understood by the team, and that the impact of changes to existing designs is addressed.

The Product Owner Role gathers User Needs as input to development, an important role in establishing Design Inputs.

Consistent with the **AGILE** principle of **DONE** is **DONE**, a **STORY** is DONE when it has been determined to meet the needs of the STAKEHOLDERs. This is an element of Design Validation. The Product Owner, representing STAKEHOLDERs, is primarily responsible for establishing the acceptance criteria for each **STORY**, including the criteria for determining whether the **STORY** meets the needs of the STAKEHOLDERs. When the **STORY** is delivered, the Product Owner Role, again representing STAKEHOLDERs, is primarily responsible for determining whether the acceptance criteria has been met.

*The Product Owner Role establishes the acceptance criteria for each **STORY** and determines whether it has been met as **STORIES** are delivered – important elements of Design Validation.*

Beyond the Design Inputs related to User Needs, the Product Owner Role must consider inputs from Risk Management, Cybersecurity, and Usability activities. (These topics are addressed further in Clause 6.3.2 "Topics related to risk management", Clause 6.3.3 "Topics related to cybersecurity", and 6.3.4 "Topics related to Usability".). These inputs might result in the creation of specific **BACKLOG** Items and might affect the Acceptance Criteria for **BACKLOG** Items. While the Product Owner might not be the expert in Risk Management and Cybersecurity and so might need help from other experts, they are responsible for ensuring Risk Management and Cybersecurity is addressed in the **BACKLOG**.

*As part of managing the **BACKLOG** and defining Acceptance Criteria, the Product Owner Role ensures that inputs from Risk Management and Cybersecurity are addressed.*

6.2.6 Scrum master role

The Scrum Master Role (an element of Scrum) is responsible for guiding the Scrum/**AGILE** team, helping it to perform at the highest level. Among its many responsibilities, the Scrum Master Role is responsible for coaching the development teams and the organization in the mechanisms of **AGILE** development, facilitating some of the events of Scrum and guiding the team on the application and evolution of its **AGILE** practices.

Going beyond the practices of **AGILE**, the Scrum Master Role should also guide the team in the application of the requirements of the organization's QMS. Quality is a responsibility of everyone on an **AGILE** team, and the **AGILE** development model encourages the development team to make decisions on how best to develop a product, including decisions on how best to apply the requirements of the QMS, so the Scrum Master should not take responsibility away from the team, but they can provide guidance on the way to apply the QMS requirements within the framework of **AGILE** practices.

For example, if the QMS establishes a "minimum" requirement (such as a minimum number of reviewers, or a minimum amount of documentation, or a minimum amount of testing), the Scrum Master might help the team determine whether and how to go beyond the minimum in some situations. Further, as described in Clause 5.4.3 Independence of Review, **AGILE** teams should determine whether those reviewing a work item have sufficient independence from those who did

the work, and the Scrum Master can help the team determine whether the **AGILE** team has enough independence to review its own work or whether another independent review might be useful.

*The Scrum Master Role should guide the team on the proper application of the Quality Management System processes within the **AGILE** framework.*

6.2.7 Development team role

The Development Team (or Dev Team) role is the third specialty role in the Scrum framework, responsible for delivering the product by delivering the Stories defined on the **BACKLOG**. They are responsible for implementing the solution according to quality standards agreed to by the team, following the practices and mechanisms agreed to by the team. The Dev Team may include specialists who can perform only certain tasks or generalists who can perform many kinds of tasks, but in any case, the Dev Team is collectively responsible for assigning tasks to individuals who are qualified to perform them.

The Dev Team role should be primarily responsible for satisfying the requirements of the organization's QMS. Though the Scrum Master role should provide guidance to the team (as describe in Clause 6.2.7), the Dev Team role is responsible for understanding the requirements of the QMS, assigning qualified people to perform the required tasks, and collectively ensuring the QMS has been followed.

The Dev Team role should be responsible for satisfying the requirements of the Quality Management System.

6.2.8 Continuous integration

Continuous Integration (a practice of Extreme Programming) is the practice of developing small components of the software and continuously integrating those components into the complete software system. The frequency of "continuous" might depend on the size and complexity of the software system being built, but ideally would mean that whenever new code is checked into the source code repository, a software BUILD is performed to confirm that the new code does not break the software.

Continuous Integration allows for varying degrees of continuous VERIFICATION, which can be a component of Design Verification activities required by 21 CFR 820.30(f). At a minimum, new code is verified to compile and BUILD into the system, passing the coding standards and other rules necessary to allow the BUILD to complete successfully. Further, tests can be run on the newly built software system to confirm the code operates correctly. These tests could be unit tests, integration tests, or software system tests (as defined by IEC 62304), or less structured "smoke tests" that give confidence that the new code meets quality expectations. In any case, Continuous Integration and the associated Design Verification activities provide fast feedback on the quality of the code, so that if errors are injected, they are discovered quickly and more easily removed.

Continuous Integration provides for opportunities for early and frequent evaluation of the developed software –Pairing

Pairing is the practice of putting two developers on one activity. This can include requirements, design, test, and coding. The original practice from extreme programming is called pair programming where code is written by two developers working together. One programmer will type in the code and the other programmer will take a different perspective and review the code to ensure that it meets the required quality attributes. The practice has expanded to include pairing on an activity where two people working together might produce a better result than one person working in isolation. While pairing can appear to bring extra costs with two persons doing the task, this can be offset by lower defect removal costs.

The benefits of pairing include:

- Improved designs–having different perspectives on a design allows alternative ideas to be debated with the best selected;
- Effective training/mentoring–for example, more senior developers can share their best design patterns with the junior engineers as a way to propagate design expertise;

— Reduce project risk of knowledge loss–designs and programming techniques are shared within the pair. If one part of the pair leaves the company, the other pair member is still available;

— Improve quality–constant review ensures less defects are inserted. For example, pair programming is at its best when the detailed design intent is reviewed as the code is written. Also, conformance to coding standards, practices, and guidelines can be assessed as the pair develops their code. Rotation of pairs can enhance a broad understanding of the code.

This practice aligns well with the regulatory perspective that emphasizes the need to verify **DESIGN OUTPUT**S, as pairing is a form of continuous VERIFICATION.

When pairing is used as a form of review, it is necessary to consider the regulatory perspective on reviews. Regulations such as FDA's QSR Section 21 CFR 820.30(f) and standards such as IEC 62304 require that manufacturers establish procedures for verifying the device design. The essential elements from these considerations include:

— Establish acceptance criteria to ensure the **DESIGN OUTPUT** meets **DESIGN INPUT** requirements;

— Define the qualifications of the reviewers appropriate to the task;

— Document the results.

It is important for each manufacturer, to define in their software development plans and procedures, the **VERIFICATION** processes used to meet these requirements. A part of a broad **VERIFICATION** process includes peer reviews that take place throughout the development process. Peer reviews can take many forms, including formal inspections, walkthroughs, and pairing.

When using pairing as a form of peer review (as part of the overall **VERIFICATION** process), it must satisfy the same requirements as any other method of peer review, addressing the considerations described above.

Define when and how pairing is used as a review technique.

Using pairing as a review technique also raises questions around independence and appropriateness of review.

When pairs work together, they bring two perspectives to the work. This independence of thought helps prevent mistakes as the work is completed. While pairing naturally brings at least one different perspective to the work, in some cases, this is not sufficient to ensure an adequate review.

For example, certain critical, complex, or safety related software may need additional perspectives to provide the diversity of knowledge and experience to ensure an adequate review. Additional perspectives can come through changing the pairing assignments or can come through supplemental peer reviews of activities and work products completed by a pair.

When defining the tasks and task assignment as part of **ITERATION** planning and daily planning, the team should consider how an activity and work product will be reviewed. Considering the question of independence and the nature of the work being reviewed, the team should decide whether pairing is a sufficient form of review. Furthermore, as the work is completed, the pair itself should consider whether they have performed a sufficiently independent review. Results from continuous **VERIFICATION** and **VALIDATION** practices will give feedback on whether pair reviews are effective.

Consider independence and criticality when determining whether pairing is sufficient for review.

The mechanisms for performing reviews, including the use of pairing, should be defined in procedures or plans.

6.2.9 Test driven development

TDD (a practice of Extreme Programming) emphasizes that tests are as important as the code itself, encouraging tests to be created and updated as the code is created and updated. Related to **AGILE'S DONE** is **DONE** concept, the code is not DONE until tests are created and passing.

This practice is particularly important to the development of medical device software because it is an element of Design Verification required by 21 CFR 820.30(f). It encourages thorough code coverage and enables early detection of potential issues, both valuable elements of an overall Design Verification strategy.

When applying **TDD**, a test management tool is essential. For large projects, there might be hundreds or thousands of tests, each of which might have many records of test execution, the volume of evidence might be overwhelming for a regulator to analyze. A robust test management tool, one that provides summary reports with useful references to details, can demonstrate control over the design verification activities.

*Use **TDD** practices as a valuable element of a Design Verification strategy.*

6.2.10 "Stop the line"

"Stop the line" is a mechanism for monitoring a process to identify abnormalities that require attention.

"Stop the line" mechanisms include:

- Monitor the development process–To find and correct problems early in the development process, it is important to have monitoring mechanisms that will identify issues to be addressed. Some examples of mechanisms include burndown charts to signal when teams are behind in velocity, BUILD times to indicate BUILD problems, regression results to indicate quality issues, and graphs of test written/tests passed to indicate issues with quality and VERIFICATION progress.

- Make information visible to the team–To provide information to the team who can act on it, it is important to provide "information radiators" that are visible to the whole team. This will help ensure their full engagement in identifying and solving the issues in a timely manner. Information radiators can be visible in project rooms for co-located teams, and electronic versions can help tie together multi-site teams.

- Act on the information–To take action to address issues, it is important to have incentives and rewards that reinforce the culture to quickly fix issues.

"Stop the line" can be a way to identify process issues that need to be addressed by a formal corrective and preventive action (CAPA) process as required by medical device regulations, such as the FDA's QSR subpart J - 820.100 and ISO 13485 Clause 8.5.

Identify and resolve issues early using the "Stop the line" practice.

6.2.11 RETROSPECTIVES/reflections

RETROSPECTIVES and reflections are practices that encourage a team to regularly inspect their performance and take action to be more effective. Focused on the same result (learning and improvement), **RETROSPECTIVES** and reflections have one subtle difference. **RETROSPECTIVES** are event based (the end of an **ITERATION, INCREMENT**, or **RELEASE** of a product, or any significant milestone) where the essential question is a more focused, "What have we learned from what just happened that will help us improve?" Reflections are not event based, instead performed periodically depending on the nature of the team, where the essential question is broader, "How are we working as a team?" These practices should be performed regularly and frequently (such as every **ITERATION**) to encourage the team to be comfortable with the practice, and to shorten the time in which the team is operating in a less than efficient manner.

Effective **RETROSPECTIVES** and reflections focus on ensuring timeliness of the improvement (shorten duration of the problem), effectiveness of assessment (the real problems are identified), and effectiveness of the improvement

(solutions are implemented and actually solve the problem). These practices create team ownership for improving how they work as a team. This results in improved team collaboration, productivity, and quality. Like “stop the line” (described above), **RETROSPECTIVES** and reflections can be mechanisms to identify process issues that need to be addressed by a formal CAPA process.

*Stimulate ownership of continuous process improvement through **RETROSPECTIVES** and reflections.*

6.2.12 Collective ownership

Collective ownership (one of the elements of Extreme Programming) is about the team taking ownership and responsibility for the quality of the delivered software product. One aspect of collective ownership is that the team, as a group owns the design, implementation, and testing of the software, in contrast to a model where a separate person or group would own design, another owns implementation, and another owns test. Another aspect of collective ownership is that any team member can make changes to any element of the design, code, and tests, in contrast to a model where only one person owns a particular piece of the software. While it is still valuable for individuals to be experts in specific elements of the software product or development process, collective ownership suggests that expertise should be distributed throughout the team.

The benefits of collective ownership include:

- Better quality–bringing different perspectives to a task improves the quality of the resulting product;
- Better team communication–having team members all participate in design, implementation and test removes communication barriers that exist when different groups own different elements;
- Learning–shared responsibility encourages team members to create learning opportunities;
- Team flexibility–shared expertise allows project teams to respond when the team makeup changes, or when new tasks are given to the team.

Regulations such as FDA’s QSR and standards such as ISO 13485 require that development teams be trained in the tasks they perform. For collective ownership to work well, the team needs expertise in software design, implementation, and testing. Beyond the need for general expertise that is necessary for any development model, **AGILE** requires some additional skills. The team must be skilled in iterative/**EVOLUTIONARY** life cycles when iterating between detailed design, implementation, and testing. The team must be skilled in **REFACTORING**, able to know when design changes are necessary and justified, and when it is time to freeze the design and move on to adding new functionality.

Ensure team members are qualified for the tasks and deliverables they own.

A concern with collective ownership could be that responsibility is not clearly defined or that ownership is diluted leading to a lack of accountability. Even though collective ownership can allow teams to be flexible in determining how tasks are performed, it is still necessary to demonstrate approval and accountability for the completion of activities and deliverables.

6.2.13 Coding standards

Coding Standards (an element of Extreme Programming) is about the team establishing guidelines that describe what “good code” means. In support of Collective Ownership described above, Coding Standards establish the common attributes of good code to be used by everyone on the team as they write and review code.

This element of **AGILE** practices is directly relevant to regulators. Though Coding Standards are not explicitly required by regulation, FDA’s General Principles of Software Validation (FDA’s GPSV) recommends having “coding guidelines.” The IEC 62304 software standard goes further by requiring manufacturers to “establish acceptance criteria for software units,” mentioning that Coding Standards are an example of acceptance criteria.

Consistent with the **AGILE** principles of continuous improvement and the focus on value-added processes, Coding Standards are expected to evolve as the development teams needs them to. **AGILE** teams recognize that a very long, detailed, and ever-growing checklist of code criteria is impractical, and may even be detrimental, because such lists can inappropriately encourage a "checklist mentality" where the goal becomes completing the checklist and providing detailed evidence, distracting the team from value-added activities of delivering high-quality code. Instead, **AGILE** teams should view Coding Standards as dynamic, adding new items to address attributes that need specific focus (such as attributes focused on safety and security, or attributes that address problems the team has encountered in their code), and removing attributes that the team has grown to apply so well that a checklist is no longer helpful. Further, with the use of automated tools (such as advanced compilers and static-analysis tools), many attributes of a Coding Standard become the tool's responsibility for checking, removing the burden of the teams having to specifically address those attributes in a code review.

Ensure team members are qualified for the tasks and deliverables they own.

6.3 Addressing other regulatory requirements in an AGILE way

This section addresses other regulatory requirements that are relevant to software development but are not covered by IEC 62304, giving guidance on how **AGILE** practices should be considered when addressing those requirements.

6.3.1 Topics related to design validation

Design Validation is required by 21 CFR 820.30(g) Design Validation and IEC 13485 Section 7.3.7 Design and Development Validation.

Design Validation as defined in 21 CFR 820.3.z(2) means "establishing by objective evidence that device specifications conform with user needs and intended use(s)". Design Validation can be simply defined as the activities that answer a simple question: "Did we make the right product?" Design Validation activities are essential to ensure that the needs and expectations of STAKEHOLDERs are being met, that the product is usable, and that the product actually does what it is claimed to do. Design Validation is usually not just a single activity of having users test the product. Especially for complex products and products with higher safety classifications, a complete and robust Design Validation strategy should be established that includes different kinds of Design Validation activities.

When developing systems, Design Validation activities could be performed at different levels of the system. For example, when software is a subsystem in a system that includes hardware and electronic subsystems, Design Validation activities could be performed on the subsystems in isolation as well as on the complete system. In this section, unless otherwise specified, the recommendations could apply to Design Validation activities at any level.

AGILE methods address Design Validation throughout the development of the product. Consistent with the **AGILE** concepts of "**DONE** is **DONE**" and continuous integration & test, consideration for what "DONE" and "tested" means should include the confirmation that the expectations of STAKEHOLDERs are being met. As shown in Figure 17, **AGILE** practices related to Design Validation can be performed at the **PRODUCT**, **RELEASE**, **ITERATION /INCREMENT**, and **STORY LAYER**.

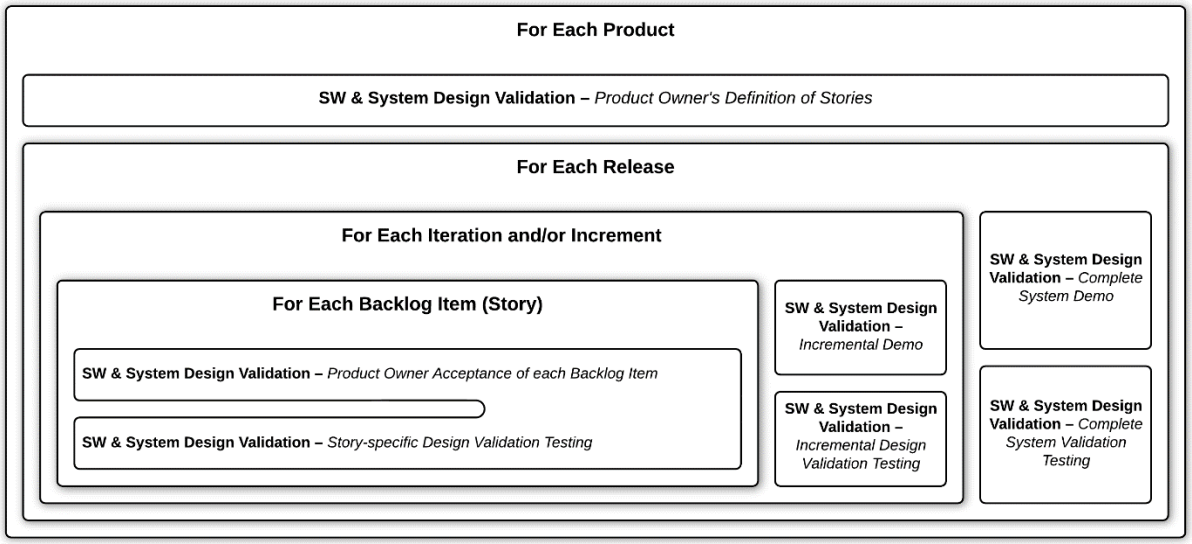

Figure 16—Design validation activities in an AGILE model

6.3.1.1 Using a STORY's definition as inputs to design validation

As described in Clause 6.1.2.1, a STORY is related to requirements in that both define what the software must do. There is a similar relationship of Stories and User Needs, with similar considerations for whether those are two different things are one combined thing.

User needs describe what users expect from the software product, such as a description of a problem to be solved, or characteristics of solution. User needs might be documented along with requirements in a requirements document or in a requirements management tool. User needs might also be described in a STORY, either as part of the description or as part of the acceptance criteria.

Just as the Agile team should decide how to address requirements and Stories (as described in Clause 6.1.2.2), they should decide whether there are two separate things for user needs and Stories, or whether user needs can be embedded into Stories. The same factors described in Clause 6.1.2.5 apply: Formality, TRACEABILITY, Document Creation, and Configuration Management.

In plans and procedures, define how Stories and user needs are used in defining the product.

Even if user needs and Stories are two separate things, the description of a STORY and/or its acceptance criteria can still provide input to design validation activities. Even if the content of the STORY does not meet all of the expectations for formality or completeness, the content of the STORY could provide useful input, such as by identifying the users who might be impacted by the STORY, providing a description of a new or modified user need, or describing a new solution to a user need.

A STORY can be an input to Design Validation activities.

6.3.1.2 Product owner role in design validation

As described in Clause 6.2.6, the Product Owner Role is responsible for establishing acceptance criteria for **STORIES** and determining whether the delivered Stories have met the acceptance criteria. While the Dev Team is focused on delivering a solution that works (is verified), the Product Owner is focused on delivering a solution that satisfies STAKEHOLDERs' expectations (is validated). As shown in Figure 17, as the Product is defined through creation of **STORIES** on the **BACKLOG**, the Product Owner Role establishes the criteria for each **STORY** by which delivered functionality will be validated. Then, when the **STORY** is delivered, the Product Owner (representing the user's perspective) determines

whether the functionality meets the criteria. To satisfy regulatory requirements and expectations for Design Validation, the Product Owner Role alone is not sufficient, but the role is a valuable element of a broader Design Validation Strategy.

The Product Owner Role's responsibility to define Acceptance Criteria and validate Stories as they are delivered are important elements of a Design Validation Strategy.

6.3.1.3 Demo as a design validation activity

As described in Clause 6.2.5, the **ITERATION** Review includes a demonstration (the **ITERATION** Demo), where each team demonstrates the changes they made to the product through the **STORIES** they delivered during the previous **ITERATION**. For larger systems built by many teams, there may be an additional Demo (called the System Demo in the Scaled **AGILE** Framework (SAFe), where the integration of many teams' features is demonstrated to STAKEHOLDERs. The main purpose of a Demo is to get feedback from STAKEHOLDERs to help answer the question: "Did we make the right product?"

While the Demo could be a formal Design Validation activity including the execution of specific tests that provide robust objective evidence, it could also be an informal activity where STAKEHOLDERs run the software and provide feedback. The scope of the demo could be a small prototype intended as a proof of concept, or it could be a complete feature that is equivalent to what will be in the final product. In any case, while it might not satisfy all regulatory expectations for Design Validation, the Demo is a valuable element of a broader Design Validation Strategy. Frequent demonstration throughout the product development life cycle allows the team to get VALIDATION feedback early when changes are easier to address and reduces the risk of significant issues being found in late-stage VALIDATION activities.

Frequent demonstration of working software (such as during **ITERATION** *Demos or System Demos) is an important element of a Design Validation Strategy.*

6.3.1.4 Design validation testing in the AGILE model

As shown in Figure 17, within **AGILE'S ITERATIVE/EVOLUTIONARY** life cycle model there are three levels where Design Validation testing activities can be performed: **STORY**, **INCREMENT**, and **RELEASE**.

At the **STORY** level, Design Validation testing can be performed as small pieces of functionality are created when the **STORY** is delivered. If relevant to the **STORY**, specific activities would be identified in the plan for delivering the **STORY**, such as a specific Design Validation test or a focused human factors evaluation.

At the **INCREMENT** level, Design Validation testing can be performed as larger pieces of functionality are created, such as the delivery of a complete feature that addresses a STAKEHOLDER's needs. Depending on the size and complexity of the product being developed, this could happen at the end of an **ITERATION** when several related **STORIES** are delivered or could be after multiple **ITERATIONS** where many **STORIES** have been delivered and a complex system or subsystem has been integrated. The Design Validation activities would be identified in the plan for delivering the Feature, such as a set of Design Validation tests or a "Formative Evaluation," as defined in the FDA guidance "Applying Human Factors and Usability Engineering to Medical Devices" and IEC 62366-1 "Medical devices – Part 1: Application of usability engineering to medical devices."

At the **RELEASE** level, Design Validation testing can be performed on the complete system that is intended for **RELEASE**. These activities can include the execution of a broad suite of Design Validation tests, a clinical study, or a "Summative Evaluation" as defined in IEC 62366-1.

As with many aspects of the **AGILE** development model, the Dev Teams have the responsibility to determine the best way to address Design Validation activities as the product is being developed, considering all three levels where those activities could take place (**STORY**, **ITERATION**/**INCREMENT**, **RELEASE**). To support this, two planning elements are necessary.

First is to have the development plan define a Design Validation Strategy that gives guidance to the teams on the kinds of activities to consider at each level, with recommendations on the scope and granularity of each activity and criteria for determining whether/when an activity would need to be redone as **INCREMENTAL** development proceeds.

Second is to require that the specific, context-dependent Design Validation activities be addressed during planning activities (such as Sprint Planning defined by Scrum, RELEASE Planning defined by Extreme Programming, or Program INCREMENT Planning defined by the SAFe).

*Planning should address **INCREMENTAL** Design Validation testing to perform for a **STORY**, **INCREMENT**, and **RELEASE**, as part of a complete Design Validation Strategy.*

Depending on the complexity of the software and system being developed, it may be difficult to do **INCREMENTAL** VALIDATION at the **STORY** level. Since Stories are often focused on a small part of functionality, a **STORY** might not deliver enough functionality from a user's perspective, so it may be difficult to determine whether the needs of the User have been met. It might be necessary to deliver a set **STORIES** to completely address a user need, and then perform design VALIDATION activities at the **INCREMENT** level to validate the new functionality added by that set of **STORIES**.

The decision of whether to perform Design Validation testing at the **STORY** level or **INCREMENT** level can also be dependent on the goal of that testing. If a goal is to validate an important small piece of functionality to get feedback to guide future development of related functionality, then **STORY** level VALIDATION testing would be useful. If a goal is to confirm that a complete set of finished functionality meets the needs of the user, then **INCREMENT** level VALIDATION testing would be useful.

Determine the appropriate level of granularity for Design Validation testing, depending on the complexity of the product and goal of the testing activities.

6.3.1.5 Aligning incremental design validation with regulatory requirements

With the **AGILE** approach of performing Design Validation throughout the development of the product there is a concern about alignment with regulations. In the requirements for Design Validation, 21 CFR 820.30(g) specifically requires Design Validation to be performed on "final production units or their equivalent", and ISO 13485 Section 7.3.7 requires Design Validation to be performed on "representative product," which includes "initial production units, batches or their equivalents."

These requirements can be misinterpreted to mean that Design Validation must be near the end of product development, and that Design Validation activities performed earlier in the development cycle do not meet the regulatory requirements. While some Design Validation activities should certainly be done near the end, deferring all activities to the end has two significant drawbacks. First, when problems are found late in development, the cost of change is higher, which might encourage an organization to accept a problem as not being worth the cost to eliminate. Second, when problems are found late in development and changes are made, there is a higher risk of introducing new errors that might not be detected.

***INCREMENTAL** Design Validation provides opportunities to find problems early, where corrections are more likely to be made with less risk of introducing other problems*

While **INCREMENTAL** Design Validation provides benefits, the requirement for "final production units or their equivalent" must still be addressed.

Clearly there is value in performing Design Validation on "final units." Variations in the production environment or changes to a product's design since it was validated could affect the answer to the question "Did we build the right product?" While some Design Validation activities must be performed on final production units, the regulatory requirements do not specify that ALL Design Validation activities be performed on final units.

There is value in getting early feedback before having final units or their equivalent. There is value in getting feedback even before a design has been created, such as through presenting ideas or simple sketches that ask STAKEHOLDERS, "If we built this, do you believe it will be the right product?" There is value in getting feedback on versions of the product what may not equivalent to the final units, such as to address specific usability issues with simple bench-built prototypes before investing in the development of a complete production-equivalent product.

The opportunity to perform Design Validation on “equivalent” units align well with **AGILE’S ITERATIVE/EVOLUTIONARY** life cycle model. Consistent with the AGILE Manifesto’s principle of “Working software is the primary measure of progress,” as the product is developed, **AGILE** teams strive to deliver a working product that, if accepted by the Product Owner and STAKEHOLDERS, could become the final product in production, so even in the early stages of development, the completed portions of the software are intended to be “production equivalent.” Further, the **ITERATIVE/EVOLUTIONARY** life cycle demands that when changes are made, teams always consider the impact on the product and activities that were performed before the change. When Design Validation activities are performed on early versions that are thought to be production equivalent, later incremental changes must always consider whether those early Design Validation activities need to be redone, or whether the elements of the software that were validated are still equivalent to what will become the final version.

A comprehensive and effective Design Validation Strategy can include the evaluation of production-equivalent units and early design elements that are not production equivalent.

6.3.2 Topics related to risk management (safety)

Regulations require medical device companies to follow a robust set of safety risk management activities in their product development. ISO 14971 provides requirements for risk management related to a medical device, and IEC 62304 Section 7 provides additional requirements related to medical device software. These activities include risk planning, risk analysis, risk control identification, and risk control verification. The documentation and approval should be in place in accordance with the organization’s quality management system. The risk analysis is on-going throughout the project so as to capture and deal effectively with any new risks that emerge. Additional guidance regarding software risk management can be found in ISO/IEC TR 80002-1:2009 “Medical device software – Part 1: Guidance on the application of ISO 14971 to medical device software.”

As described in ISO/IEC TR 80002-1:2009, risk management activities are not separate activities around the development of a software, they are an integral part of the entire software development process. While **AGILE** does not provide practices related specifically to risk planning and risk assessment, it does provide a framework for executing risk management activities.

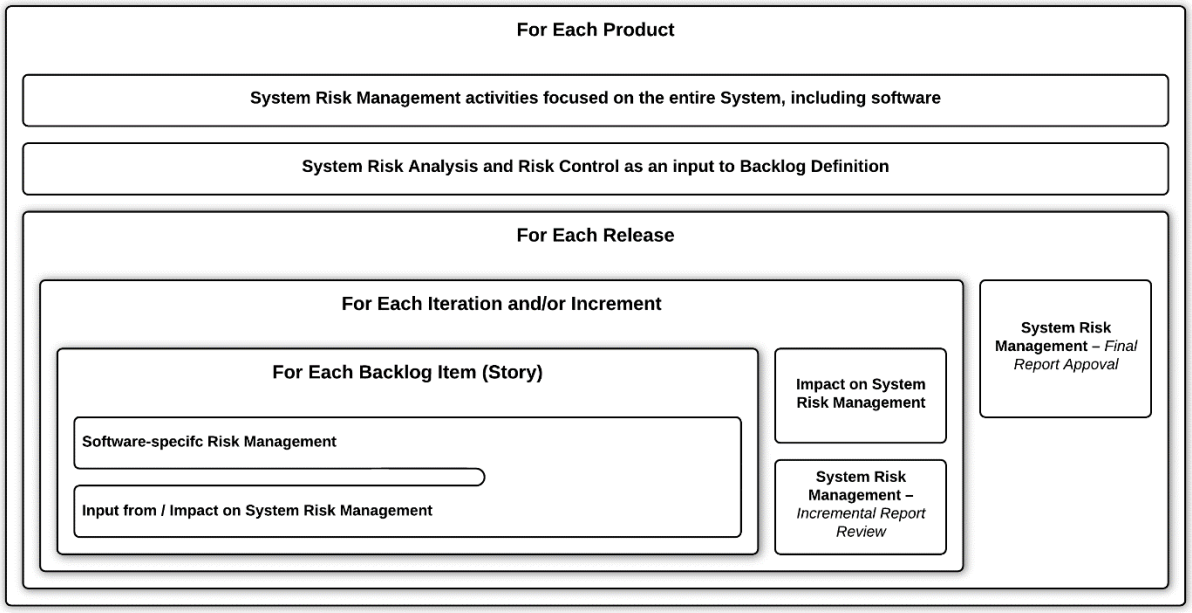

Figure 17—Risk management activities in an AGILE model

As shown in Figure 18, risk management activities should be performed at the **PRODUCT**, **RELEASE**, **ITERATION** /**INCREMENT**, and **STORY LAYER**.

6.3.2.1 Risk management as part of BACKLOG management

The activities of risk analysis and risk control relate to the **AGILE** mechanisms of **BACKLOG** management. As new features are added to the **BACKLOG** and their definition is refined, risk analysis should be performed to determine whether new hazards should be considered and whether risk estimations should be reevaluated. As risk control measures are identified they are input into the creation of new **BACKLOG** items (**STORIES**) and the refinement of existing **BACKLOG** items.

*Integrate traditional risk analysis and risk control into **STORY** creation and **BACKLOG** management activities.*

6.3.2.2 Risk management as part of BACKLOG execution

Consistent with the **AGILE** concept of "**DONE** is **DONE**," consideration for what "DONE" means should include the completion of activities related to risk management.

As **STORIES** are planned, the inputs from risk analysis and risk control are considered as part of the acceptance criteria for the **STORY**, and software-specific risk management activities (such as those identified in IEC 62304 Clause 7 and ISO/IEC TR 80002-1:2009) are planned. As **STORIES** are delivered, risk management activities are performed, and documentation is completed as relevant to that **STORY**. In the **INCREMENT** layer, as a set of related **STORIES** are completed, documentation and review activities are performed, to incrementally complete risk management activities. Finally, in the **RELEASE** layer, "**DONE** is **DONE**" ensures that all risk management activities and documents are complete and verified.

The risk management activities performed at the **STORY LAYER** are often limited to be only those activities relevant to the scope of the **STORY**, focusing mostly on the elements of the system that are impacted by the **STORY**. In contrast, the risk management activities at the **INCREMENT** and **RELEASE** LAYERS are often broad to address the entire system.

*Use **DONE** is **DONE** criteria in the **STORY**, **INCREMENT**, and **RELEASE** LAYERS to ensure that risk management activities are complete.*

As **STORIES** are planned and delivered, any new risk that is identified can be added to the **BACKLOG** for proper disposition within the risk process. At an **INCREMENT** boundary when multiple **STORIES** are planned and delivered, risks from **STORY** interactions can be identified and added to the **BACKLOG**.

*Reevaluate safety risk as **STORIES** and **INCREMENTS** are completed.*

There are some additional benefits that **AGILE** methods bring to ensure effective safety risk management. For example:

- Team collaboration practices ensures focus on safe design;
- Frequent customer interaction/**VALIDATION** provides feedback on the usability and safety of the product;
- Continuous integration gives visibility to the effectiveness of the risk control measures.

6.3.2.3 Integrating risk management expertise

Since risk management activities should be performed throughout the software development process, **AGILE** teams should address how those activities are performed and who should do them.

Risk management expertise can be represented on an **AGILE** team in two ways. First is to have that expertise in one or more people who are full-time members of the team. This is useful when the team has to address risk management during all or most of the work they do. Second is to have risk management experts ready to join a team as necessary, either as a part-time team member when the team has significant risk management issues to address, or as temporary help to the team to perform specific risk management tasks or coach the team on occasional risk management issues.

In either case, as teams plan and deliver items from their **BACKLOG**, they should determine how they apply the expertise needed to address risk management issues.

Beyond the activities owned by an **AGILE** development team, some activities might involve a cross functional team focused specifically on risk management activities. When there is significant effort needed to address risk management, and/or when risk management activities might address issues that have a broad impact on many development teams, there might be value in having a team of specialists with the expertise to do those activities effectively and efficiently. With such a specialized team, it is important to consider how that team interacts with other development teams to ensure all teams work effectively to ensure risk management is addressed throughout the development effort.

When creating **AGILE** *teams, consider how Risk Management expertise will be represented.*

6.3.3 Topics related to cybersecurity

Cybersecurity needs to be built-in to the design of the product from the very beginning. As new features are added to product, the cybersecurity implications of that feature need to be considered. For example, the introduction of a new feature, or modification of an existing feature, may result in new vulnerabilities and therefore every **INCREMENTAL** BUILD of the product should be evaluated for security. Additional guidance regarding security risk management can be found in AAMI TIR57 “Principles for medical device security – Risk management.”

While **AGILE** does not provide practices related specifically to cybersecurity, it does provide a framework for addressing cybersecurity activities throughout the development life cycle.

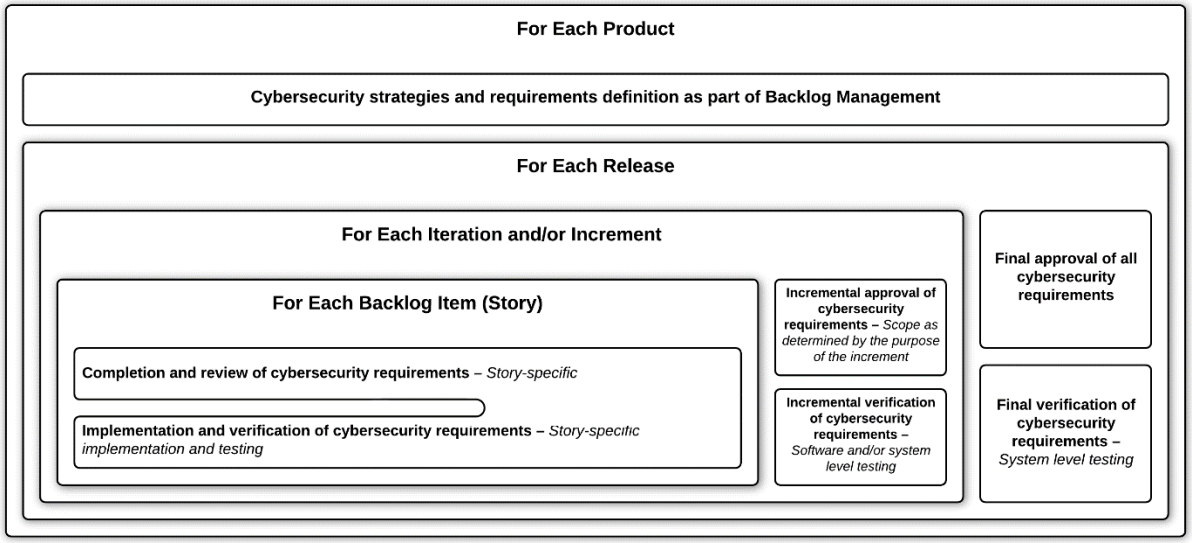

Figure 18—Cybersecurity activities in an AGILE model

As shown in Figure 19, cybersecurity activities should be performed at the **PRODUCT**, **RELEASE**, **ITERATION/INCREMENT**, and **STORY LAYER**.

6.3.3.1 Cybersecurity as part of BACKLOG management

As a product is being defined, cybersecurity strategies are established, addressing the level of concern related to security and identifying how cybersecurity will be built into the product. Those strategies might define features that should be built into the product, which would influence the structure and content of the product **BACKLOG**. As functional requirements for cybersecurity are defined, they are input into the creation of new **BACKLOG** items (**STORIES**) and the

refinement of existing **BACKLOG** items. As non-functional requirements for cybersecurity are defined, the acceptance criteria for many **BACKLOG** items might be refined.

*Cybersecurity strategies and requirements are inputs to **STORY** creation and **BACKLOG** management activities.*

6.3.3.2 Cybersecurity as part of BACKLOG execution

Consistent with the **AGILE** concept of "**DONE** is **DONE**," consideration for what "DONE" means should include the completion of activities related to cybersecurity.

As **STORIES** are planned, cybersecurity requirements might be refined as part of refining the acceptance criteria for the **STORY**. As **STORIES** are delivered, cybersecurity activities are performed, and cybersecurity is factored into the design of the product. Documentation related to cybersecurity is completed as relevant to that **STORY**. In the **INCREMENT** layer, as a set of related **STORIES** are completed, documentation and review activities are performed, to incrementally complete cybersecurity activities. Finally, in the **RELEASE** LAYER, **DONE** is **DONE** would ensure that all cybersecurity activities and documents are complete and verified.

The cybersecurity activities performed at the **STORY LAYER** are often limited to be only those activities relevant to the scope of the **STORY**, focusing mostly on the elements of the system that are impacted by the **STORY**. In contrast, the cybersecurity activities at the **INCREMENT** and **RELEASE** LAYERS are often broad to address the entire system.

*Use **DONE** is **DONE** criteria in the **STORY**, **INCREMENT**, and **RELEASE** LAYERS to ensure that cybersecurity activities are complete.*

As **STORIES** are planned and delivered, any new cybersecurity issues that are uncovered can be added to the **BACKLOG**. At an **INCREMENT** boundary when multiple **STORIES** are planned and delivered, cybersecurity issues related to **STORY** interactions can be identified and added to the **BACKLOG**.

*Reevaluate cybersecurity as **STORIES** and **INCREMENTS** are completed.*

6.3.3.3 Integrating cybersecurity expertise

A challenge that an organization might face when implementing a cybersecurity program is that the internal security expertise could be viewed as being separate from the product development team. This disconnect can lead to inefficiencies and an un-secure product; it can also lead to a slow response time to emerging threats.

Cybersecurity expertise can be represented on an **AGILE** team in two ways. First is to have that expertise in one or more people who are full-time members of the team. This is useful when the team has to address cybersecurity during all or most of the work they do. Second is to have cybersecurity experts ready to join a team as necessary, either as a part-time team member when the team has significant cybersecurity issues to address, or as temporary help to the team to perform specific cybersecurity-related tasks or coach the team on occasional cybersecurity issues. In either case, as teams plan and deliver items from their **BACKLOG**, they should determine how they apply the expertise needed to address cybersecurity issues.

*When creating **AGILE** teams, consider how cybersecurity expertise will be represented.*

6.3.4 Topics related to usability

As described in IEC 62366-1 "Medical Devices – Part 1: Application of usability engineering to medical devices," usability is an important aspect to be considered in the design of medical devices, especially as it relates to safety and risk management. While medical device regulations and quality system standards mention usability in general terms, the IEC 62366-1 standard provides specific guidance on the activities and deliverables necessary to demonstrate that usability has been sufficiently addressed.

AGILE provides some mechanisms intended to address usability, and further provides a framework for addressing usability activities throughout the development life cycle.

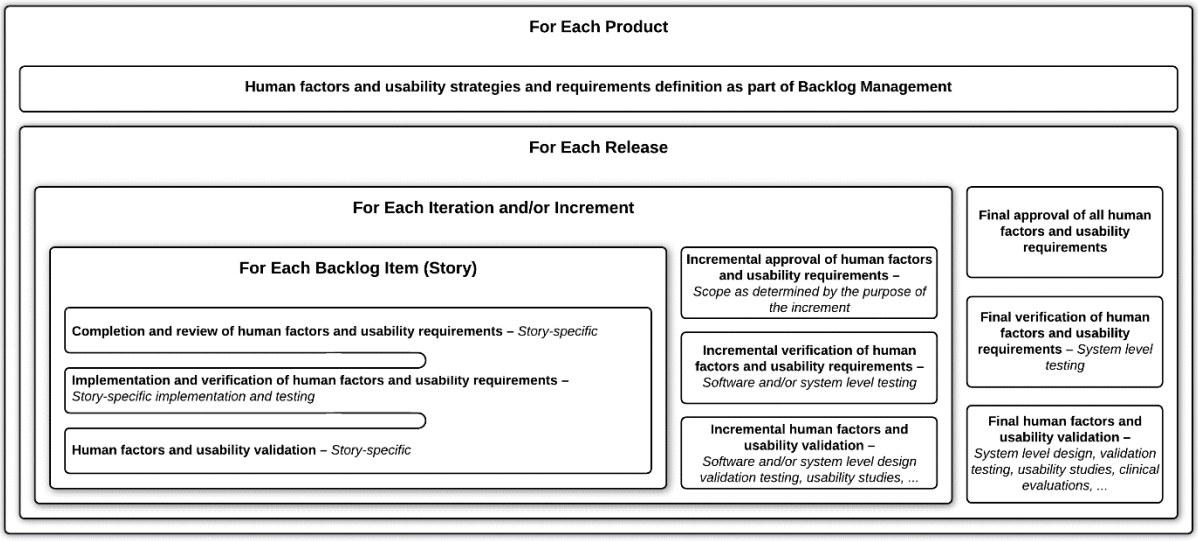

Figure 19—Usability activities in an AGILE model

As shown in Figure 20, usability activities should be performed at the **PRODUCT**, **RELEASE**, **ITERATION/INCREMENT**, and **STORY LAYER**.

6.3.4.1 Product owner's role in usability

As described in Clause 6.2.6, the Product Owner is responsible for the **BACKLOG** and responsible for determining that the delivered product meets the needs of the customers/STAKEHOLDERs. Regarding usability, this has two important elements. First, the Product Owner is responsible for addressing usability when defining the **BACKLOG**, including defining **BACKLOG** items that represent usability features that should be built into the product, and defining acceptance criteria for **BACKLOG** items related to the usability characteristics to be considered as the product is created. Second, the Product Owner is responsible for determining that the delivered product satisfies the expectations and requirement for usability.

The person in the role does not necessarily do all of this alone, they may get help from specialists in human factors and user-experience to define the **BACKLOG** items and the acceptance criteria related to usability and may get help performing VALIDATION activities that confirm usability expectations are being met.

The Product Owner role, with help as needed, is responsible for addressing usability.

6.3.4.2 Usability as part of BACKLOG management

As a product is being defined, usability strategies are established, addressing the level of importance that usability, human-factors and user-interfaces has for the product, and identifying how usability will be built into the product. Those strategies might define features or identify attributes of features that should be built into the product, which would influence the structure and content of the product **BACKLOG**. Those strategies might also identify activities related to usability that should be performed, such as user-research, user requirements elicitation, usability studies, and the Formative Evaluations and Summative Evaluations as defined in IEC 62366-1. As functional requirements for usability are defined, they are input into the creation of new **BACKLOG** items (Stories) and the refinement of existing **BACKLOG** items. As non-functional requirements for usability are defined, the acceptance criteria for many **BACKLOG** items might be refined.

Usability strategies and requirements are inputs to **STORY** creation and **BACKLOG** management activities.

While **AGILE** methods tend to discourage "big definition/design up front (BDUF)," preferring instead to have definition/design emerge iteratively, depending on the complexity of the product's user interface and the importance of usability, some "up-front" definition and user-interface design activities may be useful. **AGILE** teams should find the appropriate balance between up-front and iterative activities to ensure the product is designed with usability in mind.

Similarly, when determining the scope and timing of usability evaluations (especially Summative Evaluations), **AGILE** teams should find the appropriate balance between iterative activities that provide early and frequent feedback on usability, and "at the end" activities that demonstrate the final product has sufficiently addressed usability.

BACKLOG *management should address usability activities that should be* DONE *up-front, at-the-end, and iteratively.*

6.3.4.3 Usability as part of BACKLOG execution

Consistent with the **AGILE** concept of "**DONE** is **DONE**," consideration for what "DONE" means should include the completion of activities related to usability.

As **STORIES** are planned, usability requirements might be refined as part of refining the acceptance criteria for the **STORY**. As **STORIES** are delivered, usability activities are performed. In the **INCREMENT** layer, as a set of related stories are completed, documentation and review activities are performed, to incrementally complete activities. Finally, in the **RELEASE** LAYER, DONE is DONE would ensure that all activities and documents are complete and verified.

The activities performed at the **STORY LAYER** are often limited to be only those activities relevant to the scope of the **STORY**, focusing mostly on the elements of the system that are impacted by the **STORY**. In contrast, the activities at the **INCREMENT** and **RELEASE** LAYERS are often broad to address the entire system.

Use **DONE** *is* **DONE** *criteria in the* ***STORY, INCREMENT,*** *and* ***RELEASE*** *LAYERS to ensure that usability activities are complete.*

As **STORIES** are planned and delivered, any new usability issues that are uncovered can be added to the **BACKLOG**. At an **INCREMENT** boundary when multiple **STORIES** are planned and delivered, usability issues related to **STORY** interactions can be identified and added to the **BACKLOG.**

Reevaluate usability as ***STORIES*** *and* ***INCREMENTS*** *are completed.*

6.3.4.4 Addressing usability during product demos

As described in Clause 6.2.5, the **ITERATION** Review includes a demonstration (the **ITERATION** Demo), where each team can get feedback from STAKEHOLDERs. As further described in Clause 6.3.1.2, the **ITERATION** Demo or the broader System Demos are opportunities for Design Validation activities to take place. Similarly, these demonstrations are opportunities to get feedback on usability. During these demos, human factors and user-experience specialists can seek informal feedback on usability as the product is being developed, as an input to the design of more rigorous and structured usability activities such as a usability study or Formative Evaluation.

ITERATION *Demos and System Demos can be used to get feedback on usability.*

6.3.4.5 Integrating usability expertise

For large systems and those where usability is of high importance, specific expertise in human factors and user-experience might be needed to ensure that usability is sufficiently addressed throughout the definition and implementation of a product.

Usability expertise can be represented on an **AGILE** team in two ways. First is to have that expertise in one or more people who are full-time members of the team. This is useful when the team has to address usability during all or most of the work they do. Second is to have usability experts ready to join a team as necessary, either as a part-time team member when the team has significant usability issues to address, or as temporary help to the team to perform specific usability-related tasks or coach the team on occasional usability issues. In either case, as teams plan and deliver items from their **BACKLOG**, they should determine how they apply the expertise needed to address usability issues.

Beyond the representation on **AGILE** teams, there may be value in having a team focused specifically on usability activities. When there is significant effort needed to understand the expectations and requirements for usability, and/or when there is significant effort needed to evaluate the product for usability (such as usability testing or design validation), there might be value in having a team of specialists with the expertise to do those activities effectively and efficiently. With such a specialized team, it is important to consider how that team interacts with other development teams to ensure all teams work effectively to ensure usability is addressed throughout the development effort.

When creating **AGILE** *teams, consider how usability expertise will be represented.*

Appendix A
(informative)

Analysis of the value statements from the manifesto for AGILE software development

In Clause 4.1, the AGILE Perspective, the value statements from the Manifesto for AGILE Software Development were described along with a brief description of how they can align with the values of the regulatory perspective. The following sections provide a deeper analysis to give more insight on how the AGILE Manifesto can be appropriately applied to medical device software development.

A.1 Individuals and interactions over process and tools

This value statement recognizes that skilled people working well together will produce good software, and that processes and tools are useful but insufficient. While processes and tools are necessary and useful, they cannot by themselves ensure that a high quality product is produced. Effective processes and tools will help a good team perform even better, but no amount of processes and tools will help a poor team perform well.

Safe and effective medical device software cannot be designed from a script or a checklist. While it is necessary to have an environment where design controls exist, it is also necessary to allow engineers and designers to think, to create, to solve problems, and implement great solutions.

A criticism of this value is that it can lead to a lack of discipline. If this value were taken to an extreme, that criticism might be valid, but when applied with balance, a better discipline can result.

Documented processes and supportive tools bring discipline to a development process by codifying procedures and behaviors that have been deemed important. This kind of discipline is especially useful to establish common rules for consistency, to give guidance for training requirements, and to ensure that essential processes/practices are made visible and not forgotten or taken for granted. This kind of discipline also has some drawbacks. Problems can occur if the team does not feel responsibility for the process, resulting in processes that are misunderstood or ignored, or if the documented processes do not support execution realities, resulting in the “two sets of books” mentality where the things we say do not match what we actually do.

AGILE imposes a different focus to discipline, the discipline of an interactive team of individuals who are aligned on shared goals, shared values, and shared principles. Following the principles of “inspect and adapt” and “visibility,” an AGILE team is obligated to regularly ask itself how the processes and practices are working, and then make improvements. This kind of discipline is especially useful to establish good rules and guidance that adapts to the uniqueness of the current context, to establish expectations for team behavior that leads to continuous improvement, and to ensure that the documented process matches the execution reality. This kind of discipline also has some drawbacks. Problems can occur if the team does not accept responsibility for its processes and practices, if the team is not sufficiently trained on fundamental concepts for software development processes and the requirements of a quality management system, or if schedule pressures discourage the team from committing the necessary time to the inspect-and-adapt principle.

A potential way to address this is to apply both kinds of discipline. Apply the discipline of a clear and sufficient documented process to establish the rigor necessary for medical device software. Apply the discipline of AGILE to adapt a rigorous process to the team’s context and focus on continuous improvement. This combination can result in good alignment between the AGILE and regulatory perspectives on individuals and interactions, process, and tools. Both perspectives place value on a development team that takes ownership of the software it creates, and the processes and practices used to create it.

Augment the discipline of necessary robust processes and tools with the discipline of a team that is responsible for its processes and practices.

A.2 Working software over comprehensive documentation

This value statement recognizes that working software is the ultimate deliverable, the best indicator of progress and the best indicator that the needs of the customer are being satisfied. While documentation is necessary and useful, it is of little value without working software to go with it. Clear and sufficient documentation helps the development team ensure they have a complete and good product, and it helps external STAKEHOLDERs evaluate the completeness and goodness of what they receive, but no amount of documentation will compensate for an incomplete or deficient product.

Regulations and standards require certain kinds of documentation to provide objective evidence that your development process has been followed to develop a safe and effective product. **AGILE** should not be used as a way to subvert required documentation. Saying "we are **AGILE**, we don't need design documentation" is not a defendable position in the medical device software world. More importantly, effective documentation can help a software team produce a safe and effective product. **AGILE** should not be used as a reason to diminish the value of documentation. Instead, **AGILE** concepts should be applied to ensure that valuable documentation is produced, and wasteful documentation is eliminated.

A criticism of this value is that it can be seen as being contrary to a common principle in the medical device software world: "If it isn't documented, it didn't happen." While this is a useful notion to highlight the importance of producing objective evidence, the over-emphasis of it can lead to a checklist mentality where documentation is produced simply to satisfy some real or imagined requirement while providing no value to the development team. **AGILE** challenges development teams to produce documentation only when it provides value from a user centered point of view, be it business, regulatory or end user.

When assessing the value of documentation, the development team should consider what is valuable to them and what is valuable to regulators. The most valuable documentation will satisfy both, and not be viewed as favoring one to the detriment of the other.

Produce documentation that is valuable to both the development team and regulatory STAKEHOLDERs.

For more information about documentation, see Clause 5.5.

A.3 Customer collaboration over contract negotiation

This value statement recognizes that collaboration with a customer who is engaged in the development process is more likely to deliver software that meets the customer's needs. While contracts are useful, they cannot by themselves ensure that a development team will produce a product that satisfies the customer. Clear and sufficient contracts are useful to establish common understanding and to set reasonable expectations, while inflexible and adversarial contracts can inhibit adaptation and stifle innovation that is essential to new product development.

Customer contracts, in terms of the legally binding documents and the processes to create them, can be an important element of running a business, but they are not of significant interest to regulatory agencies. Quality system regulations and guidance do not address the process of creating contracts between manufacturers and customers, so in that regard, this value statement demands no unique interest from the medical device software world.

More important to regulatory agencies and manufacturers is the process for identifying the needs of customers and patients. Understanding user needs and the intended use of the software product and ensuring that the software is designed and validated for that use, is a vital element of creating a safe and effective product. From this point of view, **AGILE'S** emphasis on customer collaboration aligns very well with the regulatory perspective's emphasis on software **VALIDATION**. Several **AGILE** practices demand that a customer be intimately involved with the development team to define requirements for the software, to guide changes and further elaboration of requirements as development proceeds, and to define and declare acceptance as functionality is delivered. In support of **AGILE'S INCREMENTAL/EVOLUTIONARY** life cycle, the customer is involved throughout the project to perform regular and frequent **VALIDATION** at many levels, from the small bits of functionality created via **STORIES**, to the large delivery of a complete solution via a **RELEASE**.

A challenge presented by this value statement is in defining who the customer is and how the collaboration will occur. Medical device software is often intended to be used by many customers, often in many different contexts. Furthermore, medical device software is often part of a larger system, which brings even more diversity in the number of customers and in the way they will interact with the software. It can be a challenge for the Customer Role on an **AGILE** team to represent multiple viewpoints of that diverse customer community.

Consistent with **AGILE'S INCREMENTAL/EVOLUTIONARY** life cycle, the emphasis on customer collaboration means that the software's definition will evolve over time. While some high-level definition will be completed to varying degrees before the development team begins, most of the detailed definition and even some of the high-level definition will emerge as the software development work proceeds. This may be uncomfortable for manufacturers and regulators who are more familiar with "big definition up-front" models where definition work is more complete before software development begins. The **AGILE** perspective recognizes that for most development projects the full definition cannot be known up front and the most responsible course of action is to capture the definition, requirements, and specifications that are known in advance and to then fully elaborate them over the whole life cycle of the project. It is critical for the project's development plan to explain how the definition of the software will emerge and explain how artifacts will be created to demonstrate how that definition emerges in a controlled way.

For more information about emerging definition and design, see Clause 5.1, 6.3, and 6.1.4.

*Close collaboration with customers and **EMERGENT** definition mechanisms encourage the development of software that satisfies the customer's needs and expectations.*

A.4 Responding to change over following a plan

This value statement recognizes that change is inevitable in new product development and should be embraced as a good and useful thing. While plans are necessary and useful, they cannot, by themselves ensure a good product will be delivered. Clear and sufficient plans help establish reasonable expectations, set the project up for success, and provide a means of control, but no amount of detail in a plan can predict and control the dynamic nature of new product development.

Early criticisms of **AGILE** suggested that **AGILE** did not place value on planning, or that **AGILE** teams would not commit to plans. This was a gross misunderstanding of the principles and practices of **AGILE**. To the contrary, **AGILE** puts tremendous emphasis on planning, with varying levels of planning happening every day during a project. For example, high level planning occurs at the **PRODUCT LAYER** during the creation of a product **BACKLOG** and **RELEASE** definition. Mid-level planning occurs for an **INCREMENT** to determine the functionality to be created in the **INCREMENT**. Low-level planning occurs for the daily tasks to be completed. Plans are created and frequently adjusted with full participation of customers, project managers and the development team. The important point of this value statement is that plans must not be seen as rigid mandates for how work must be performed, but instead should be used as effective guidance that can be adapted as work proceeds.

The **AGILE** and regulatory perspectives align very well on the importance placed on planning, but they might not align on the expectations for the frequency and the nature of changes to plans. **AGILE'S** emphasis on frequent planning activities may not align with regulatory STAKEHOLDERs who are more comfortable with linear development models where planning occurs up-front, followed by execution and control of the plan. This difference should not be exaggerated, however, since even linear development models prescribe the need to update and adjust plans as work proceeds. To reconcile any potential misalignment, it is important to define the planning activities that occur, and the objective evidence that is produced. Furthermore, it is important to have feedback mechanisms in place that will indicate the difference between controlled and robust responses to change, versus uncontrolled and risk-filled thrashing.

Define the planning mechanisms and establish feedback mechanisms to demonstrate effective control of changes.

For more information about managing changes, see Clause 5.6 and 6.1.9.

Appendix B
(informative)

Applying AGILE development to IEC 62304 – Quick Guide

Over the past several years, **AGILE** software development has become an accepted method for developing medical device software products.

This section provides simplified guidance on developing medical device software while conforming to IEC 62304 and other guidance/standards.

B.1 Regulation, Standards and Procedures

Regulation and standard documents describe activities that a software team should perform, such as product definition and requirements specification, high-level design, software coding, and testing. They provide requirements that define the work that must be completed but leave it to the manufacturer to define "how" they will do the work which is defined in the manufacturer's procedures and processes.

The use of **AGILE** for the development of medical device software brings some challenges, in particular creating the documentation that conforms to regulation, standards and the manufacturer's procedures.

An issue with using **AGILE** in a regulated environment is that many **AGILE** development tools and **AGILE** Lifecyle management tools do not support the automated assembly of required Medical Device documentation deliverables. For regulators or auditors who are more comfortable with linear life cycle models, an **AGILE** life cycle might not meet their expectations unless the documentation is presented in a way that the auditor is used to seeing.

To avoid misunderstandings, incorrect assumptions, or differing expectations, it is important for software development plans to define the life cycle model to be used and to define how they will demonstrate compliance to IEC 62304 and other quality, regulatory and standards requirements.

B.2 AGILE Software Development and 62304

Software development consists of a well-established set of activities that are defined in IEC 62304. These activities can be executed using many different life cycle models. IEC 62304, Section B.1.1 states that "62304 does not require a particular Software Development Life Cycle Model".

Some development models as defined at ISO/IEC/IEEE 12207 include: Incremental, Spiral, Iterative, Evolutionary, **AGILE**. When using **AGILE** to develop medical device software it is important to define how the requirements of 62304 will be met.

The following two items are important regardless of the SOFTWARE DEVELOPMENT LIFE CYCLE MODEL used:

- Demonstrating Conformance to Guidance and Standards: You are required to complete all activities as defined by applicable guidance, standards and manufactures procedures.
- Automating Conformance: With the complexity of the requirements and standards today, it's quite difficult to comply with guidance and standards using manual methods. As much as possible, teams should look to automate the objective evidence of conformance to guidance and standards. Teams should purchase tools that support an **AGILE** software development method and that automate the creation/update of deliverables required by guidance and standards.

Figure B.1 below shows a high-level view of IEC 62304 requirements throughout the development life cycle.

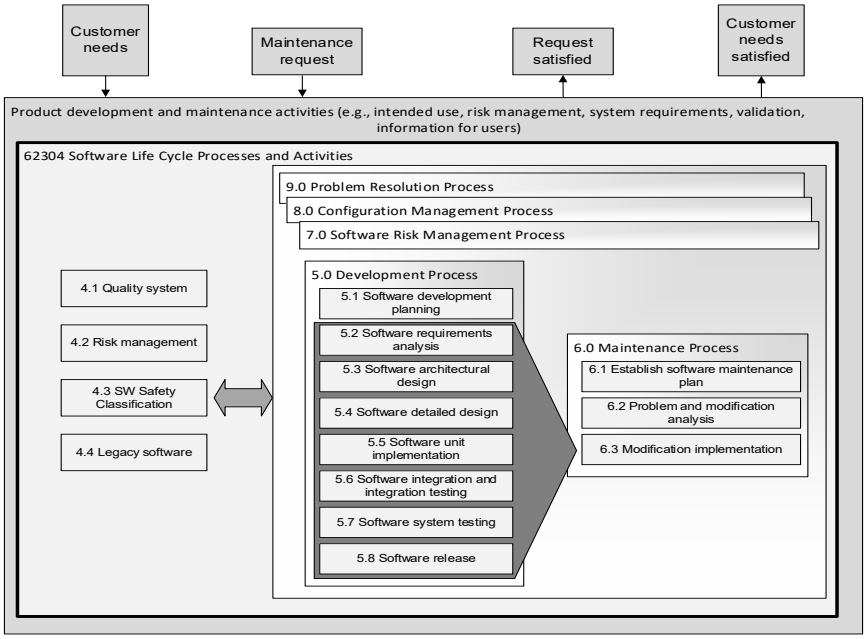

Figure B.1 - Overview of software development processes and activities

In Figure B.2, the 4 layers of abstraction are shown with an example of IEC 62304 activities in each layer. As part of the development planning activities (IEC 62304 Clause 5.1), you will define how you will meet the requirements of IEC 62304 and other applicable guidance and standards.

AGILE's life cycle delivers new functionality small bits at a time through **STORIES**, **INCREMENTS**, and **RELEASES**. **AGILE** emphasizes the concept of **DONE IS DONE**, where all activities and documentation for a piece of functionality are completed as the functionality is delivered, leaving little or no work to be completed later.
AGILE teams define "**DONE**ness criteria" that address all activities related to definition, implementation, and testing.

The development of objective evidence that demonstrates conformance to applicable guidance and standards should be included as part of the "Definition of **DONE**" for each **AGILE** layer.

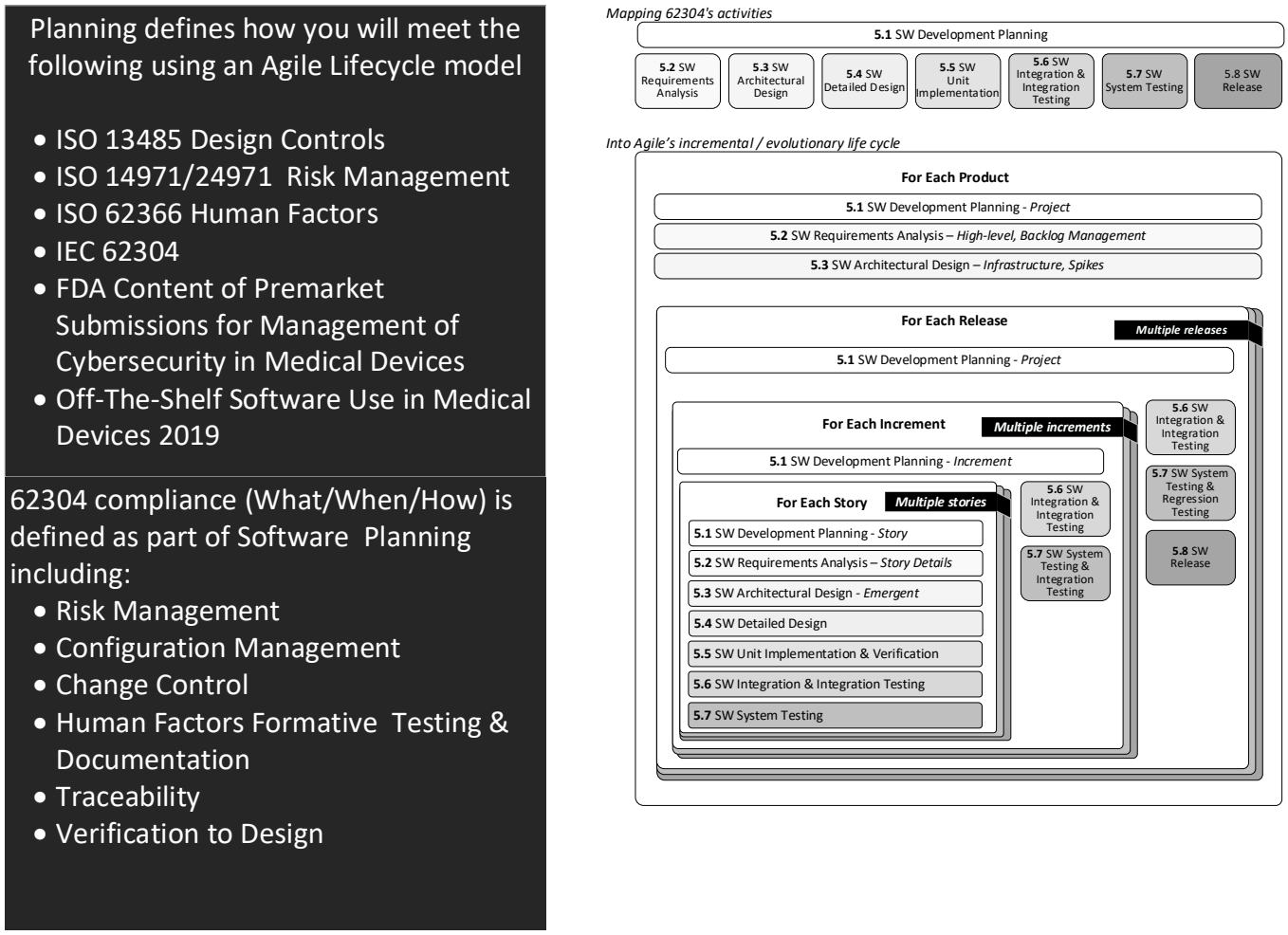

Figure B.2 – Medical Device Standards and 62304 processes

With respect to software or product development, the term **AGILE** does not denote a specific methodology, approach, or practice (i.e., there is not a generally accepted, specific **AGILE** methodology) but rather an umbrella term that is typically applied when software or product development.

- fits with the spirit of the Manifesto for **AGILE** Software Development
- is more empirical than deterministic, and
- is **EVOLUTIONARY** and frequently iterative.

B.3 AGILE - 4 layers of abstraction

The **AGILE** life cycle is delineated into the following 4 layers.

- The **PRODUCT LAYER**
- The **RELEASE LAYER**
- The **INCREMENT LAYER**
- The **STORY LAYER**

B.3.1 Planning for Development for each of the AGILE Layers

AGILE development planning governs the activities conducted throughout the **AGILE** process. Planning establishes the framework of how the **AGILE** life cycle will be implemented in the **AGILE** layers.

To avoid misunderstandings, incorrect assumptions, or differing expectations, it is important for software development plans and procedures to define the life cycle to be used. Planning should address the following:

—

- The process activities to be executed.
- The timing and sequencing of those activities.
- The inputs used and the outputs generated by an activity.
- The process artifacts (the audit trail) to be produced.
- How the life cycle integrates with risk management.
- How change management and configuration management are implemented.
- Determination and setting of software safety class.
- TRACEABILITY to/from: requirements, risk (security and safety), VERIFICATION and VALIDATION.

IEC 62304 has a specific required set of planning activities that must be performed based on your software safety class. Figure B.3 shows the IEC 62304 planning activities per clause. Initial planning usually occurs outside the start of **AGILE** development.

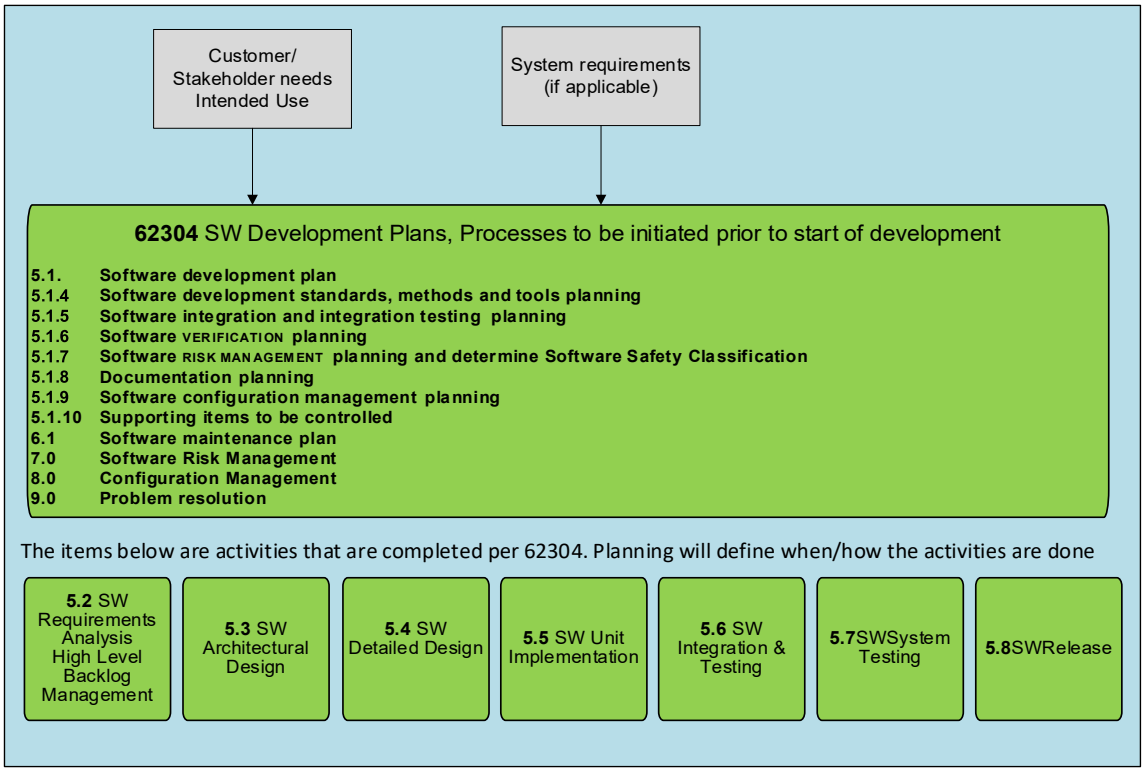

Figure B.3 – 62304 Planning activities

- For the **PRODUCT LAYER**:
  - planning addresses the high-level activities of project management: scoping the project, forming teams, and gathering resources, planning the long-range duration and cost, and

organizing the project into **RELEASE**s including maintenance RELEASES. The **PRODUCT LAYER**:consists of the complete set of activities related to the development and delivery of a software product.

- For the **RELEASE LAYER**:
  - planning addresses the mid-level activities of project management: scoping the **RELEASE**, establishing high- level integration points of the software and other subsystems, scheduling at the middle level, and organizing the **RELEASE** into **INCREMENT**s.

- For the **INCREMENT LAYER**:
  - planning addresses the low-level activities of project management: planning and scheduling at the low level, establishing low-level integration points within software subsystems or other subsystems, and organizing an **INCREMENT** into **STORIES**.

- For the **STORY LAYER**:
  - planning addresses detailed team and individual planning: assigning tasks, planning execution details for daily tasks, and tracking progress.

The planning will also cover how the development teams will provide objective evidence of TRACEABILITY per IEC 62304, ISO 14971 and other guidance and standards as applicable.

B.3.2 The **PRODUCT LAYER**

The **PRODUCT LAYER** consists of the complete set of activities related to the development and delivery of a software product.

The following shows the procedures and planning documents to be created in the **PRODUCT LAYER** or prior to the start of **AGILE** software development. This is not the only way to show conformance to required medical device deliverables but gives an example of an approach that can be used successfully.

Activities at the **PRODUCT LAYER** may consist of (includes compliance activities):

- Establishment of a risk management process in accordance with ISO 14971 that incorporates software risk management as outlined in IEC 62304 section 7 – enabling re-assessment of risk as new software requirements are established and supporting software architecture emerges.
- Establishment of configuration management procedures for controlling the planned software artifacts. These may be an existing part of your quality management system.
- Establishment of a problem resolution procedure for addressing defects in verified and RELEASED software. This may be an existing part of your quality management system.
- Establishment of a product risk management file and initial safety risk analysis including assignment of software safety class.
- Establishment of a software development plan incorporating the elements defined in IEC 62304 section 5.1
  - Software development standards, methods and tools planning
  - Software integration and integration testing planning
  - Software **VERIFICATION** planning
  - Software risk management (Safety and Security/Cybersecurity)
  - Documentation planning
  - Software configuration management planning
  - Supporting items to be controlled
  - TRACEABILITY as required by 62304and other guidance/standards.

B.3.3 The RELEASE LAYER

The **RELEASE LAYER** consists of activities to create a usable product.

Depending on planning, a **RELEASE** might generate a product that could be shipped to the field, or it might generate a completed, consistent set of functionalities intended only for internal use.

Planning at the **RELEASE LAYER** addresses scoping the **RELEASE**, establishing high-level integration points of the software and other subsystems, and organizing the **RELEASE** into **INCREMENT**s. For long RELEASE cycles, a **RELEASE** can be made up of one or more **INCREMENT**s as defined in the development plan.

In the **RELEASE LAYER** as complete sets of functionalities are integrated with other complete sets or as the software is integrated with other subsystems, that integration is tested. The Integrations, and what is integrated at what time is defined as part of the project planning process.

Activities in the **RELEASE LAYER** should confirm that conformance to applicable guidance and standards as defined in development planning are complete.

Activities at the **RELEASE LAYER** may consist of (includes compliance activities):

- Establishment of a RELEASE plan, identifying the scope, VERIFICATION points and deliverables.
- Following completion of planned **INCREMENT**s, establishment of any additional software integration tests and software requirements tests not able to be completed at the **STORY** or **INCREMENT LAYERS**. This should include evaluation of test procedures.
- VERIFICATION that all documentation of TRACEABILITY of requirements, risk management, human factors, and cybersecurity were completed.
- Requirements, as they are elaborated must trace to from user needs to the end state of the fully elaborated requirement.
- Full TRACEABILITY from the elaborated requirements, to risks (both safety and security) and to testing as required by IEC 62304.
- Full Software Integration and Requirements Testing – including a documented record of results (this includes tests established during the development of earlier **RELEASE**s).
    - Note that throughout **AGILE** software development, integration and requirements testing at the **STORY** and **INCREMENT LAYERS** and for earlier **RELEASE**s may have occurred, but that does not guarantee that when all software is fully integrated that testing will still succeed. The act of integrating all software may surface issues that were not seen earlier. This is easily facilitated by the AGILE use of full automated testing.
    - If full software integration and requirements testing is not performed, then a regression analysis should be considered as appropriate to establish regression testing required.

B.3.4 The INCREMENT LAYER

The Increment Layer is composed of activities to create a set of useful functionalities, although not necessarily a complete software product. A **RELEASE** is made up of one or more **INCREMENT**s. The timeframe and scope associated with an **INCREMENT** is defined as part of development planning.

Activities at the **INCREMENT LAYER** may consist of:

- Following completion of all planned **STORIES**, may include the establishment of any additional software integration tests and software requirements tests not able to be completed at the **STORY LAYER**.

B.3.5 The STORY LAYER

The **STORY LAYER** is composed of activities to create a small piece of functionality, although not necessarily a complete set. The timeframe associated with a **STORY** is defined as part of development planning.

Planning at the **STORY LAYER** addresses detailed team and individual planning: assigning tasks, planning execution details for daily tasks, and tracking progress.

Activities at the **STORY LAYER** may consist of (includes compliance activities):

- Creating/updating requirements specification(s) for the part of the software in scope of the **STORY** with TRACEABILITY to requirement source (e.g., system, user and risk specifications). This could include review and approval.
-
- Creation/Updating TRACEABILITY between the requirements and software specifications are created. Software requirements analysis does not happen in the **INCREMENT LAYER** or **RELEASE LAYER**, although the planning activities of these layers will address which requirements are within the scope of the **RELEASE**S, **INCREMENT**s, and **STORIES**.
- Using the **AGILE** concepts of simple design and **REFACTORING**, the creation/updating of software architecture emerges (including OTS Software/SOUP), with TRACEABILITY to Requirements, as the architecture emerges to support the new functionality being added by a STORY. This could include review and approval
- Creation/Updating Software detailed design and software unit implementation and **VERIFICATION**. Using the concepts of simple design and **REFACTORING**, detailed design emerges as the code is developed. Using the concepts of **TDD**, unit-level tests are designed and executed as the code is developed.
- Using the concepts of continuous integration, new functionality is integrated into the broader software product as the code is developed. Using the concepts of **TDD**, that integration is tested as the pieces are put together.
- As a **STORY** is further elaborated, establishment of requirements test(s) (or other VERIFICATION method) – including evaluation of test procedures and TRACEABILITY to requirements. Depending on what is needed to execute that test, the test might be executed in the **STORY LAYER** or it might not be executed until required elements of the system are integrated in the **INCREMENT LAYER** or **RELEASE LAYER**.
- Assessment of any risks (safety and cybersecurity) associated with the further elaborated requirements and architectural design established for the **STORY**. Identified risks are managed as appropriate in the security file and risk management file.

B.4 Software Integration and Integration Testing

Software integration and integration testing may happen in any of three layers:

- In the **STORY LAYER**, using the concepts of continuous integration, new functionality is integrated into the broader software product as the code is developed. That integration of the **STORY LAYER** is tested as the **STORIES** are put together.
- In the **INCREMENT LAYER**, as individual **STORIES** are completed to create a complete set of functionalities, that integration is tested as the code is developed.
- In the **RELEASE LAYER**, as complete sets of functionalities are integrated with other complete sets or as the software is integrated with other subsystems, that integration is tested.
    - Throughout **AGILE** software development, there may be integration testing at the STORY, INCREMENT, and RELEASE LAYERS, but that does not guarantee that when all of the software is fully integrated, that the testing completed during INCREMENT testing is sufficient to find all issue with the fully integrated software. The act of final integration of the software may surface issues that were not seen before.
    - You may take credit for testing that is completed, but once all incremental RELEASES are completed, the need for a Full Integration test of all incremental RELEASES should be evaluated.

- The approach to Full Integration and Testing should be defined as part of the planning activities.

B.5 Software System Testing

Software testing also happens in any of the three layers of abstraction.

In the **STORY LAYER**, as a requirement is defined, a test (or other **VERIFICATION** method) is defined to verify it. Depending on what is needed to execute that test, the test might be executed in the **STORY LAYER** or it might not be executed until required elements of the system are integrated in the **INCREMENT LAYER** or **RELEASE LAYER**.

In the **INCREMENT LAYER**, tests defined by a **STORY** might be executed or new tests might be created to verify requirements related to subsystem integrations.

In the **RELEASE LAYER**, existing tests might be re-executed, or new tests might be created to verify system-level integration requirements.

Appendix C
(informative)

Quick reference guide

As mentioned in Clause 1.3, Organization: Navigating this document, some readers of this TIR may be interested in a subset of the topics covered by this TIR. The following table is intended to help readers quickly find guidance regarding the activities of interest.

Table C.2—Index of activities of interest

Clause	Title	Suggestions on adopting an agile approach	Conceptually, how is agile compliant to regulations?	What are the details of demonstrating compliance?
4	Setting the stage			
4.1	The **AGILE** perspective	X		
4.2	The regulatory perspective		X	
4.3	Aligning perspectives		X	
5	Aligning on concepts	X		
5.2	Inputs and outputs	X		
5.3	**DESIGN INPUTS** and **DESIGN OUTPUTS**	X		
5.4	Design reviews	X		
5.5	Documentation	X		
5.6	Managing the dynamic nature of **AGILE**	X		
5.7	Human safety risk management	X		
6	Aligning on practices	X		
6.1	Addressing IEC 62304 in an **AGILE** way	X		X
6.2	Using **AGILE** practices for regulatory compliance		X	X
6.3	Addressing other regulatory requirements in an **AGILE** way		X	X
Appendix A	Analysis of the value statements from the Manifesto for **AGILE** software development		X	
Appendix B	Applying **AGILE** development to IEC 62304 – A Quick Guide	X		X

Bibliography

[1] ANSI/AAMI/IEC 62304:2006+Amd1:2015, *Medical device software—Software life cycle processes.* American National Standards Institute, New York, NY. Association of the Advancement of Medical Instrumentation, Arlington, VA. International Electrotechnical Commission, Geneva, Switzerland; 2006

[2] ISO 14971:2019, *Medical devices – Application of risk management to medical devices*. International Organization for Standardization, Geneva, Switzerland; 2019

[3] ISO/IEC 12207:2008*, Systems and software engineering—Software lifecycle processes.* International Organization for Standardization, Geneva, Switzerland. International Electrotechnical Commission, Geneva, Switzerland; 2008

[4] ISO/IEC 20926:2009, *Software and system engineering—Software measurement—IFPUG functional size measurement method*. International Organization for Standardization, Geneva, Switzerland. International Electrotechnical Commission, Geneva, Switzerland; 2009

[5] ISO/IEC 20000-1:2011, *Information technology—Service management—Part 1: Service management system requirements*. International Organization for Standardization, Geneva, Switzerland. International Electrotechnical Commission. Geneva, Switzerland; 2011

[6] IEEE/ISO/IEC 24765:2017, *Systems and software engineering vocabulary*. Institute of Electrical and Electronics Engineers, Piscataway, NJ. International Organization for Standardization, Geneva, Switzerland. International Electrotechnical Commission. Geneva, Switzerland; 2017

[7] IEEE 828-2012, *IEEE Standard for configuration management in systems and software engineering*. Institute of Electrical and Electronics Engineers, Piscataway, NJ; 2012

[8] Mountain Goat Software. *User Stories*. Available at: http://www.mountaingoatsoftware.com/topics/user-stories. July 3, 2012

[9] Food and Drug Administration (FDA), 21 U.S.C. Ch. 9 § 301 et seq. Federal Food, Drug, and Cosmetic Act (FD&C Act); 1938

[10] Food and Drug Administration (FDA), 21 CFR 820 Quality System Regulation (QSR); 2018

[11] Food and Drug Administration (FDA), *Guidance Document, General Principles of Software Validation; Guidance for Industry and FDA Staff*; January 2002

[12] Food and Drug Administration (FDA), *Guidance for the content of premarket submissions for software contained in medical devices*

[13] Smartsheet.com. *The Ultimate Agile Dictionary*. Available at: https://www.smartsheet.com/ultimate-agile-dictionary. Accessed July 3, 2012

[14] Cockburn A. *Agile Software Development*. Reading (MA): Addison-Wesley, 2002

[15] Cohn M. Available at: https://agilemanifesto.org/ Accessed July 3, 2012

[16] McConnell S. *Rapid development*. Redmond (WA): Microsoft Press, A Division of Microsoft Corporation, 1996. Copyright ©1996 Steve McConnell. All rights reserved. Used with permission