ECSS-E-HB-40-01A
7 April 2020

Space engineering

Agile software development handbook

ECSS Secretariat
ESA-ESTEC
Requirements & Standards Division
Noordwijk, The Netherlands

Foreword

This Handbook is one document of the series of ECSS Documents intended to be used as supporting material for ECSS Standards in space projects and applications. ECSS is a cooperative effort of the European Space Agency, national space agencies and European industry associations for the purpose of developing and maintaining common standards.

The material in this Handbook is defined in terms of description and recommendation how to organize and perform the work of space software engineering and product assurance following an Agile approach.

This handbook has been prepared by the ECSS-E-HB-40-01A Working Group, reviewed by the ECSS Executive Secretariat and approved by the ECSS Technical Authority.

Disclaimer

ECSS does not provide any warranty whatsoever, whether expressed, implied, or statutory, including, but not limited to, any warranty of merchantability or fitness for a particular purpose or any warranty that the contents of the item are error-free. In no respect shall ECSS incur any liability for any damages, including, but not limited to, direct, indirect, special, or consequential damages arising out of, resulting from, or in any way connected to the use of this document, whether or not based upon warranty, business agreement, tort, or otherwise; whether or not injury was sustained by persons or property or otherwise; and whether or not loss was sustained from, or arose out of, the results of, the item, or any services that may be provided by ECSS.

Published by: ESA Requirements and Standards Division
ESTEC, P.O. Box 299,
2200 AG Noordwijk
The Netherlands
Copyright: 2020© by the European Space Agency for the members of ECSS

Change log

ECSS-E-HB-40-01A 7 April 2020	First issue

Table of contents

Introduction ... 9

1 Scope ... 10

2 References ... 11

3 Terms, definitions and abbreviated terms ... 12

3.1 Terms from other documents ... 12

3.2 Terms specific to the present document ... 12

3.3 Abbreviated terms ... 16

4 Introduction to the Agile software development approach ... 18

4.1 Introduction to Agile ... 18

4.1.1 General ... 18

4.1.2 Agile characteristics (as derived from the manifesto) ... 19

4.1.3 Lean management ... 21

4.2 General issues implementing Agile ... 22

5 Guidelines for Agile life cycle selection ... 25

5.1 Selecting Agile ... 25

5.2 Analysis of key factors for Agile selection ... 25

5.2.1 General ... 25

5.2.2 Customer context ... 27

5.2.3 Supplier context ... 28

5.2.4 Project context ... 28

5.2.5 Team context ... 30

5.2.6 Key Factors Summary ... 31

5.3 Agile assessment process ... 32

5.4 Selecting agile or waterfall ... 33

6 Reference models for Scrum-like Agile software life cycle ... 35

6.1 Introduction ... 35

6.2 Roles and competences ... 35

6.2.1 Overiew ... 35

6.2.2 Scrum master....35
6.2.3 Product owner....36
6.2.4 Development team....36
6.2.5 SCRUM team....37
6.2.6 Agile coach....37
6.2.7 Training and competencies....37
6.3 Exemplary Agile activities....38
6.3.1 Distinction between meeting or activity....38
6.3.2 Planning I – What will be delivered....39
6.3.3 Planning II – How will it be delivered....39
6.3.4 Sprint backlog management....40
6.3.5 Product backlog refinement....40
6.3.6 Progress tracking....41
6.3.7 Product backlog update....41
6.3.8 Coding, testing and documenting....41
6.3.9 User feedback....42
6.3.10 Review preparation....42
6.3.11 Sprint review....42
6.4 Meetings....43
6.4.1 Daily meeting....43
6.4.2 Management meeting....43
6.4.3 Retrospective....43
6.5 Organising the Agile activities and meetings in a project to create a life-cycle compliant to ECSS-E-ST-E-40....44
6.5.1 Preliminaries....44
6.5.2 Product releases....45
6.5.3 Start of the project: Sprint#0....45
6.5.4 Development phase: Sprints #1 - #N....46
6.5.5 Acceptance phase....46
6.6 Software lifecycle definition....47
6.6.1 ECSS-E-ST-40 reviews....47
6.6.2 Organising the ECSS-E-ST-40 reviews in an Agile software approach....48
6.6.3 Selecting the right model....57

7 Guidelines for software project management....58

7.1 Introduction....58
7.2 Software Project Management approach....58
7.2.1 Overview....58

7.2.2 Management objectives and priorities 58
7.2.3 Schedule management 62
7.2.4 Assumptions, dependencies and constraints..64
7.2.5 Work breakdown structure 65
7.2.6 Roles...65
7.2.7 Risk management 66
7.2.8 Monitoring and controlling mechanisms 67
7.2.9 Staffing Plan...71
7.2.10 Software procurement process...73
7.2.11 Supplier management 73
7.3 Software development approach 74
7.3.1 Strategy to the software development 74
7.3.2 Software project development lifecycle 74
7.3.3 Relationship with the system development lifecycle 74
7.3.4 Reviews and milestones identification and associated documentation 74
7.4 Software engineering standards and techniques 74
7.5 Software development and software testing environment 74
7.6 Software documentation plan 75

8 Guidelines for software engineering processes 76

8.1 Overview 76
8.2 Software related system requirements process 76
8.3 requirements and architectural engineering 76
8.3.1 Software requirements analysis 76
8.3.2 Software architectural design 82
8.4 Software design and implementation engineering...84
8.5 Software validation 86
8.6 Software delivery and acceptance 87
8.7 Software verification 89
8.8 Software operations..92
8.9 Software maintenance 92
8.9.1 Overview...92
8.9.2 Agile maintenance challenges...92
8.9.3 Tailoring Agile to Maintenance 93
8.10 Independent software verification and validation...96

9 Guidelines for software product assurance and configuration management 97

9.1 Software product assurance 97

9.1.1 Introduction 97
9.1.2 Planning of software product assurance activities 98
9.1.3 Software product assurance reporting 98
9.1.4 Technical Debt and noncompliance of Quality Requirements 99
9.1.5 Software criticality 100
9.1.6 Software problem management 100
9.1.7 Control of non-conformances 101
9.1.8 Software development environment aspects 101
9.1.9 Summary of software product assurance activities in Agile 102
9.2 Software configuration management 103
9.2.1 Introduction 103
9.2.2 Agile software configuration management challenges 103
9.2.3 Agile methods for configuration management 105
9.2.4 Summary of software configuration activities in Agile 106

Figures

Figure 4-1: From Plan-driven approach to Value-driven approach 20
Figure 4-2: The Lean Thinking House (for details see LEAN-PRIMER) 22
Figure 5-1: Factors for adopting Agile process 26
Figure 5-2: Agile selection factors scale 27
Figure 6-1: Organisation of activities during a sprint 38
Figure 6-2: Exemplar Agile lifecycle 44
Figure 6-3: Model 1: Review driven lifecycle 52
Figure 6-4 Model 2: More flexible review driven lifecycle 54
Figure 6-5: Review driven lifecycle with full flexibility 55
Figure 6-6: Sprint driven lifecycle with formalisation 56
Figure 7-1: Project Management Triangle 59
Figure 7-2: Cost Management: change for free 60
Figure 7-3: Sample Burndown Chart for a Sprint 63
Figure 7-4: Example for an Agile work breakdown 65
Figure 7-5: Agile model supports risk management 66
Figure 7-6: Success of continuous integration tests 69
Figure 7-7: A team metric dashboard 69
Figure 7-8: Summary of performed work 70
Figure 7-9: Delivered business value in a project 71
Figure 8-1: Example of User Story and Tasks 78
Figure 8-2: Kano model showing means to ensure customer satisfaction 80

Tables

Table 5-1: Supplier context 28
Table 5-2: Project context 29
Table 5-3: Team context 30
Table 5-4: Key factors for selection of classical or agile lifecycle 31
Table 5-5: Aspects for the selection of agile or waterfall approach 33
Table 6-1 – Overview of the different models 50
Table 6-2 – Examples of selection of models based on project characteristics 57
Table 8-1: Mapping ECSS-E-ST-40 to Agile activities. Software Requirements Analysis 81
Table 8-2: Mapping ECSS-E-ST-40 to Agile activities. Software architectural design 83
Table 8-3: Mapping ECSS-E-ST-40 to Agile activities. Software Detailed Design, Coding and testing, and Integration 84
Table 8-4: Mapping ECSS-E-ST-40 to Agile activities. Software validation 86
Table 8-5: Mapping ECSS-E-ST-40 to Agile activities. Software Delivery and Acceptance 88
Table 8-6 Mapping ECSS-E-ST-40 to Agile activities. Software Verification 90
Table 8-7: Mapping ECSS-E-ST-40 to Agile activities. Software Maintenance 95
Table 9-1: Mapping ECSS-Q-ST-80 to Agile activities. Software product assurance 102
Table 9-2: Mapping ECSS-M-ST-40 to Agile activities. Software Configuration Management 106

Introduction

The ECSS-E-ST-40 Space Engineering Software Standard defines the principles and requirements applicable to space software engineering. ECSS-E-ST-40 is always complemented by the ECSS-Q-ST-80 Space Product Assurance Standard, which specifies the product assurance aspects. This ECSS-E-HB-40-01 handbook provides more detailed guidelines and advice for adopting an Agile software development approach in space projects where ECSS-E-ST-40 and ECSS-Q-ST-80 are applicable.

1 Scope

This Handbook provides recommendations for the implementation of an Agile approach in space software projects complying with ECSS-E-ST-40 and ECSS-Q-ST-80.

This handbook is not an Agile development book, though it provides an Agile reference model based on Scrum and also covers other major Agile methods and techniques. Scrum has been selected as reference because of its widespread application in industry and its flexibility as a development framework to introduce or merge with other Agile methods and techniques. In relation to the ECSS-E-ST-40 and ECSS-Q-ST-80, this handbook does not provide any tailoring of their requirements due to the use of the Agile approach, but demonstrates how compliance towards ECSS can be achieved. This handbook does not cover contractual aspects for this particular engineering approach, although it recognises that considering the approach of fixing cost and schedule and making the scope of functionalities variable, the customer and supplier need to establish specific contractual arrangements. Furthermore, it does not impose a particular finality for the use of Agile, either as a set of team values, project management process, specific techniques or supporting exploration by prototypes.

This handbook, covers, in particular, the following:

- In clause 4, the fundamentals and principles of Agile. It also describes major Agile methods and general issues of implementing an Agile approach.
- In clause 5, the criteria for selecting an Agile lifecycle.
- In clause 6, a reference process model based on Scrum to be used to map its elements to relevant clauses of ECSS-E-ST-40.
- In clause 7, guidelines for software project management, providing advice for ECSS-E-ST-40 clause 5.3 considering the reference process model based on Scrum.
- In clause 8, guidelines for software engineering processes, providing advice for ECSS-E-ST-40 clauses 5.2, and 5.4 to 5.10, considering the reference process model based on Scrum.
- In clause 9, guidelines for software product assurance and software configuration management, providing general advice for the implementation of ECSS-Q-ST-80 and ECSS-M-ST-40 with an Agile approach.

Individual agile practices, introduced in this HB, can also be taken on-board in other software development life-cycles.

2 References

ECSS-S-ST-00-01	ECSS system - Glossary of terms
ECSS-E-ST-40	Space engineering - Software
ECSS-E-HB-40	Space engineering - Software engineering handbook
ECSS-M-ST-10	Space project management - Project planning and implementation
ECSS-M-ST-40	Space project management - Configuration and information management
ECSS-M-ST-80	Space project management - Risk management
ECSS-Q-ST-80	Space product assurance - Software product assurance
ECSS-Q-HB-80-04	Space product assurance - Software metrication programme definition and implementation handbook
Agile Manifesto	Beck, K., et al.: Agile Manifesto and Twelve Principles of Agile Software (2001). http://agilemanifesto.org
ISO/IEC 26515:2011	Systems and software engineering - Developing user documentation in an Agile environment
LEAN-PRIMER	Craig Larman and Bas Vodde. 2009. Lean Primer. Available at: http://www.leanprimer.com/downloads/lean_primer.pdf
Agilealliance	https://www.agilealliance.org
SCRUM	https://www.scrum.org

3
Terms, definitions and abbreviated terms

3.1 Terms from other documents

a. For the purpose of this document, the terms and definitions from ECSS-S-ST-00-01 apply, in particular the following terms:

1. **process**
2. **product**

b. For the purpose of this document, the terms and definitions from ECSS-E-ST-40 apply, in particular the following term:

1. **critical software**

3.2 Terms specific to the present document

3.2.1 Burndown chart

A chart that records project status, usually showing tasks completed versus time and against total number of tasks

[ISO/IEC 26515:2011]

3.2.2 Capacity

Total number of available hours for a sprint. The capacity is the available hours calculated based on resources planned holiday and company holiday if any.

3.2.3 Continuous integration

development practice according to which the source code in development is uploaded into a shared repository regularly to allow automated build, tests and quality control tools to detect and raise issues early to the development team

3.2.4 Cycle time

time elapsed between the start of the work on a particular task and its completion.

3.2.5 Daily stand-up

short daily team meeting where the team members share their project activities, synchronize themselves and identify and solve impediments that hamper them from being productive

NOTE 1 It is also known as daily meeting, daily Scrum or roll-call.

NOTE 2 Duration of a “Daily stand-up” is about 15 minutes.

3.2.6 Definition of done (DoD)

list of activities and criteria to be achieved to declare an element of the backlog as complete

NOTE 1 This element can be defined at user story, epic, feature, sprint and product release levels

NOTE 2 Elements of the DoD can be, for example: writing source code, update specification, design and user documentation, execution of unit testing, achieving a certain level of test coverage, establish compliance to a certain level of coding rules.

[adapted from the Agile Alliance]

3.2.7 Definition of ready

list of criteria that a user story has to meet to be accepted into the upcoming sprint or iteration

[adapted from the Agile Alliance]

3.2.8 Epic

big user story that is too big to be estimated. It is usually too big for an iteration and will be broken down into smaller user stories.

NOTE It is usually too big for an iteration and will be broken down into smaller user stories.

3.2.9 Feature

functional or non-functional distinguishing characteristic of a system, usually an enhancement to an existing system

[ISO/IEC 26515:2011]

3.2.10 Increment

sum of all the Product Backlog items completed during a sprint and the value of the increments of all previous sprints

3.2.11 Iteration

See “sprint”[3.2.22].

3.2.12 Lead time

time elapsed between the customer request for an increment and its delivery

3.2.13 Load factor

KPI measuring how many person days it takes to complete a story point

NOTE A story point is a relative measurement unit used to compare relatively user stories, epics (e.g. one ideal working day).

3.2.14 Pair programming

practice where two developers share the same development environment, where the first developer writes the code and the second reviews the code simultaneously and thinks ahead

NOTE Both change roles often so that the strong points of both developers can be utilised and knowledge is shared in the team.

3.2.15 Peer review

practice where code written by a developer is reviewed by another developer of the same team

3.2.16 Product backlog

ordered list of everything that is known to be needed in the product

NOTE 1 It is the single source of requirements for any changes to be made to the product.

NOTE 2 A Product Backlog is never complete. The earliest development of it lays out the initially known and best-understood requirements. The Product Backlog evolves as the product and the environment in which it will be used evolves. The Product Backlog is dynamic; it constantly changes to identify what the product needs to be appropriate, competitive, and useful. If a product exists, its Product Backlog also exists.

NOTE 3 The items can be, for examples, features, requirements, software problem reports or technical tasks.

NOTE 4 The backlog is the primary point of entry for knowledge about requirements, and the single authoritative source defining the work to be done.

[adapted from the Agile Alliance]

3.2.17 Product Owner

entity as close as possible to the end customer and users and as knowledgeable as possible to the business and solution context

NOTE See also clause 6.2.3.

3.2.18 Product Backlog Refinement

formal or informal meeting or activity to refine the backlog

NOTE 1 Examples of refinement are:

- removing user stories that no longer appear relevant,
- creating new user stories in response to newly discovered needs,
- re-assessing the relative priority of user stories, assigning estimates to user stories which have yet to receive one,
- correcting estimates in light of newly discovered information, splitting user stories which are high priority but too coarse grained to fit in an upcoming iteration.

NOTE 2 For detailed information about “Product backlog refinement” see clause 6.3.5.

[adapted from the Agile Alliance]

3.2.19 Sprint backlog

set of product backlog items selected for the sprint, plus a plan for delivering the product increment and realizing the sprint goal

3.2.20 Release plan

plan to deliver an increment to the product with a release estimated date and providing for each related sprint, an outline that specifies its content

3.2.21 Retrospective

meeting that is held at the end of an iteration with the objective of improving the process by reviewing what happened in the last iteration and identifying the good things done in it and what could have been done better in order to identify improvement actions for next iterations

NOTE The sprint retrospective occurs after the sprint review and prior to the next sprint planning.

3.2.22 Sprint

short time frame, in which a set of software features is developed, leading to a working product that can be demonstrated to stakeholders

NOTE Also known as **iteration**.

[ISO/IEC 26515:2011]

3.2.23 Sprint review

review that is held at the end of the sprint to inspect the increment and adapt the product backlog if needed

3.2.24 Stakeholder

entity who have a vested interest in the success of the project, but do not belong to the core team

3.2.25 Technical debt

metaphor for the efforts and costs that built up when code is developed without applying the optimal solution, therefore being more difficult to maintain over time

3.2.26 Test driven development

programming practice where test cases are specified before the software code is written

3.2.27 User story

story that consists of one or more sentences in the language of the end user that captures what a user does or needs to do as part of his or her job function.

NOTE It captures the 'who', 'what' and 'why' of a requirement in a simple, concise way, often limited in detail by what can be hand-written on a small paper notecard.

A good scheme for creating user stories and other backlog items is INVEST:

I Independent: The user story is self-contained, in a way that there is no inherent dependency on another user story

N Negotiable: User stories are not explicit contracts and should leave space for discussion.

V Valuable: A user story must deliver value to the stakeholders.

E Estimable: Possibility to estimate the size of a user story.

S Small: The size of User stories is such that is does not become impossible to plan/task/prioritize with a certain level of accuracy.

T Testable: The user story or its related description provides the necessary information to make test development possible.

INVEST is often used as part of the Definition of Ready for a user story.

3.2.28 Velocity

number of story points delivered/demo in a sprint. The velocity is the measure of the amount of work a team can tackle during a single sprint

The velocity is the measure of the amount of work a team can tackle during a single sprint. The velocity is calculated at the end of the sprint by adding up the points for all fully completed user stories

[adapted from SCRUM]

3.3 Abbreviated terms

For the purpose of this document, the abbreviated terms from ECSS-S-ST-00-01, ECSS-E-ST-40 and the following apply:

Abbreviation	Meaning
AR	acceptance review
CDR	critical design review
CM	configuration management
DDR	detailed design review
DoD	definition of done
DRD	document requirements definition
ECSS	European Cooperation for Space Standardization
FDD	feature driven development
HMI	human machine interface
ICD	Interface control document
IDD	ideal developer day
ISVV	independent software verification and validation
KPI	Key performance indicator
MoM	minutes of meeting
MoSCoW	Must Should Could Won't
N/A	not applicable
NCR	Non-conformance report
OBS	organisation breakdown structure
OBSW	on-board software
PBI	product backlog item
PBS	product breakdown structure
PDR	preliminary design review
QA	Quality assurance
QR	qualification review

Abbreviation	Meaning
RB	requirement baseline
SDD	Software design document
SDLC	software development lifecycle
SDP	software development plan
SLA	service level agreement
SPA	Software product assurance
SPR	Software problem report
SRS	software requirements specification
SW	software
SWRR	software requirements review
TDD	test driven development
TS	technical specification
WBS	work breakdown structure
XP	extreme programming

4
Introduction to the Agile software development approach

4.1 Introduction to Agile

4.1.1 General

Agile is an iterative, time-boxed approach to software development in which a software product is built through an evolutionary process characterised by early and frequent deliveries of product increments, intensive customer collaboration and adaptive planning and goals, with the aim of responding to change in a rapid and flexible manner.

Agile development is inspired by the Agile Manifesto, which identifies four basic values:

- **"Individuals and interactions** *over* Processes and tools
- **Working software** *over* Comprehensive documentation
- **Customer collaboration** *over* Contract negotiation
- **Responding to change** *over* Following a plan"
  [Agile Manifesto]

Each statement asserts that what is in the sentence on the left side (in bold) is valued more than what is on the right side.

Behind the Agile Manifesto lie the twelve principles, which teams working in an Agile environment follow

1. *"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
4. *Business people and developers work together daily throughout the project.*
5. *Build projects around motivated individuals. Give them the environment and support their need, and trust them to get the job done.*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*

7. *Working software is the primary measure of progress.*

8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*

9. *Continuous attention to technical excellence and good design enhances agility.*

10. *Simplicity--the art of maximizing the amount of work not done--is essential.*

11. *The best architectures, requirements, and designs emerge from self-organizing teams.*

12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly."*

[Agile Manifesto]

NOTE Clause 5 provides concrete models for different approaches towards adopting the change of requirements late in the lifecycle in compliance with ECSS-E-ST-40.

At the core of Agile software development lies the high quality of deliverables, as a result of continuous product and process improvement. Agile teams develop a working system where they optimise the way they communicate and organise their work in order to achieve the highest value. There is a strong link with Lean management as detailed in clause 4.1.3.

4.1.2 Agile characteristics (as derived from the manifesto)

Agile development methodologies heavily rely on customer collaboration and feedback and therefore offer the chance to examine the direction in which a project is heading while it is ongoing. Compared to waterfall methodologies, where product development depends on the initial requirements and testing is done at the end of the implementation, in an Agile context teams continuously review and improve requirements, design, implementation and testing throughout the life cycle. In waterfall models, by the time the project is finished it can occur that the needs of the business have changed – this fact is embraced in Agile, by providing a framework where the scope can vary and planning is adaptive.

There are different Agile methods or frameworks that can be applied, such as Scrum, Kanban, Extreme Programming, or Feature Driven Development. Irrespective of which method is chosen, all of them are *Agile*, hence highly adaptable to support the need of the project and company, and they have the following common characteristics:

- They are iterative and evolutionary. The software product is developed through repeated cycles in a rapid feedback loop between customer and technologists. The customer adaptively specifies the requirements for the next iteration based on the observations on the evolving software product.
- Analysis, design, coding and testing are continuous activities.
- The iteration length can be pre-determined (time-boxed) and the scope is fixed for each iteration. One of the main differences of Agile compared to other iterative methods is that the length of each iteration is much shorter: typically two to four weeks.
- They support adaptive planning. Agile methods are intended for situations that foresee changes during the development time and the final delivery. It can be possible that requirements change, technology changes and staff changes.
- They recognise the importance of people as the most influential factor of the project. The self-organized team plays an important role delivering working software as a primary measure of

success. Agile methods focus on the competence and motivation of the software engineers as well as the organisation, communication and management.

- They attempt to reduce resource-intensive intermediate artefacts. Resources are prioritised to produce working software from the very beginning, in some cases it can be deployed directly into operation.

Agile emphasizes the value of better dealing with justified or valuable change. Addressing the changes (e.g. to the requirements, design or implementation) is in effect a change to the scope of the project. Having three parameters: Scope, Schedule and Cost, the impact of the change can be either handled by impact on the cost and schedule or by adapting the scope accordingly. Agile lifecycle provides the methodology for handling the changes to the scope in a fixed cost and schedule frame (see Figure 4-1). The details of how changes to the schedule, cost and scope can be handled contractually are out of the scope of this Handbook.

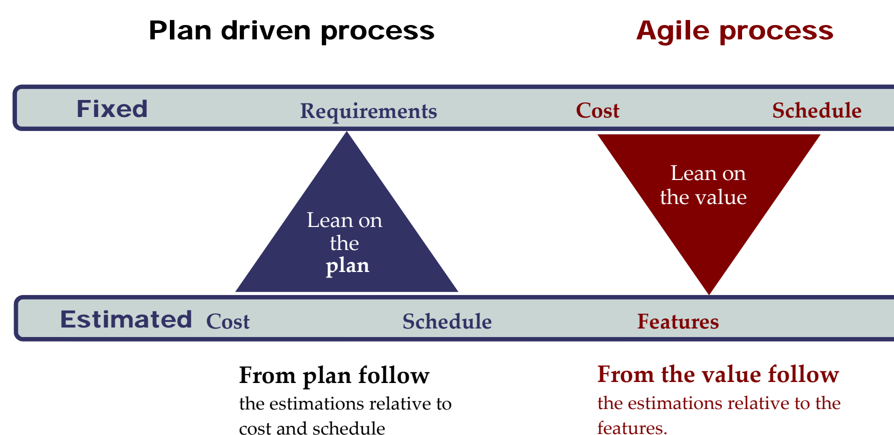

Figure 4-1: From Plan-driven approach to Value-driven approach

While defining what Agile is, it is equally important to emphasize what it is *not*. Agile is not:

- A silver bullet. Agile cannot solve all development problems and the methods are adapted to the needs of the organisation.
- Anti-documentation. Documentation is just another valuable deliverable that can be estimated and prioritised like other user functionalities. Compliance to standards is also possible. An important note is that in an Agile context classical documents are converted into "living documents" (evolving documents) that are updated on an ongoing basis.
- Anti-planning. Agile methods involve a lot of planning (daily meetings, iteration and release planning meetings); therefore it is not advised to roll out an Agile project solely on the basis that it cannot be planned.
- Anti-architecture. An architecture needs to exist before starting the project, otherwise the lack of such an artefact indicates that the user needs are still unknown and further clarification is needed (i.e. the user needs are not clear). The architecture needs to be updated regularly as it evolves while the project is running.

4.1.3 Lean management

The core idea of Lean Management is to maximise customer value while minimising waste, where waste is any unprofitable action. In simple terms, a lean approach aims at creating more value for customers with fewer resources. An organisation that follows the lean principles understands customer value and focuses its key processes to continuously increase it. The ultimate goal is to provide perfect value to the customer through a perfect value creation process that has zero waste. The idea of providing customer value relates directly to the Agile principles.

Figure 4-2 shows the principles of Lean, also applicable to Agile. The layout follows the major structure of a house and is as follows:

- The roof defines the goal of a project, i.e. to quickly deliver things of value, while still achieving the highest quality. This can include focusing on low cost approaches, safety issues, and delighting the customer.
- The foundation (bottom) relies on management that thinks, applies, and teaches lean thinking. This includes implementing long-term philosophies and basing decisions on this.
- The left pillar is on respecting people by showing lean leadership. This includes being open by providing information and showing transparency, being fair, speaking feelings, telling the truth, showing consistency, and keeping commitments.
- The right pillar is on continuous improvement, articulated by applying retrospectives for teams, visual management, and working towards flow. The figure contains more techniques like Go See, where the responsible directly go where a problem occurred to see the problem and potential corrective actions. Kaizen is also shown and describes continuously improving development processes and projects. This can be done with several means like asking 5 times why, looking for non-value adding (NVA) activities, limiting work in progress (WIP), removing multitasking, and eliminating wishful thinking.
- The middle consists of product development and 14 lean principles. Product development lists that great engineers and mentoring between junior and senior engineers are needed. Teams shall be cross-functional and use visual management techniques. Set-based concurrent engineering is a means towards agile ways of working. The 14 principles include Agile-like principles like deciding as late as possible, the creation of flow, the idea of pull, and knowledge management.

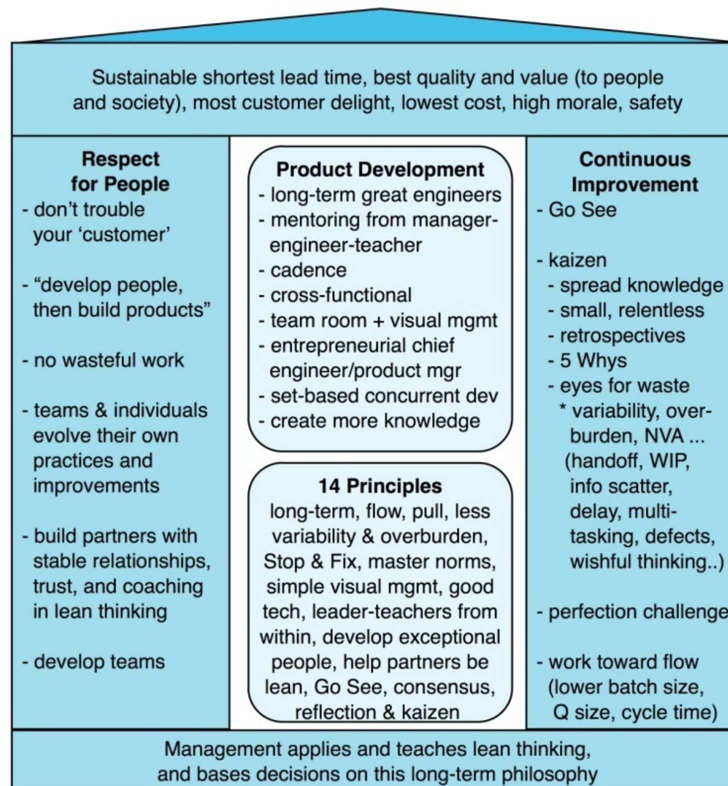

Figure 4-2: The Lean Thinking House (for details see LEAN-PRIMER)

4.2 General issues implementing Agile

One of the objectives of this handbook is to highlight the challenging points of implementing a project that follows an Agile software development process and to provide in this context best practices to common questions that can arise.

The following list comprises a number of risks, issues, perceptions and concerns with regards to the application of Agile which are addressed in this handbook.

a. Control over budget and schedule:

- Customer concerns:
  - How do we ensure that at the end of the project we do not get less than originally expected due to the changes introduced as part of the Agile process (avoid running out of effort before all required features are implemented)?
  - How do we ensure that certain functionality is available at pre-defined milestones and the schedule does not slip due to the frequent changes as part of the Agile process?
- Supplier concerns:
  - Does Agile mean that we, as suppliers, have to implement much more than originally planned within the same fixed effort and schedule?
  - Agile does not mean that the overall project schedule is flexible and extendable.
  - When team members, who are each an expert of a particular aspect of the system, work in parallel on multiple projects, how can a long term planning be achieved if the team members are reassigned to different Agile projects due to the introduced changes?

b. Scope of documentation:

- Customer Concerns:
  - When the value of the documentation is as high as the value of the working software, adopting Agile processes does not automatically mean “less documentation”.
- Supplier Concerns:
  - How do we avoid that we are asked to follow both sequential and iterative Agile processes at the same time and produce exhaustive documentation according to a sequential process logic (at the corresponding milestones) while at the same time following an iterative Agile life cycle for the development?

c. Approach towards Quality, Validation and Verification:

- Customer Concerns:
  - There is a common perception that Agile works fine for prototype software but not for operational software such as critical on-board software. Can our suppliers use Agile and still deliver critical software?
  - There is a concern that in the end, adopting Agile can mean getting software, which has not been fully qualified (validation and verification) due to lack of time.
- Supplier concerns:
  - If we follow an Agile approach while adhering to ECSS-E-ST-40 and ECSS-Q-ST-80, do we expect that each sprint is treated as a formal review, where each software release goes through the heavy validation and qualification process?
  - How do we produce the artefacts and evidences required for verification and validation of the full software system at each major joint review milestone, while in an Agile process the focus of each iteration is on producing working software for the features of that iteration?

d. Process Management Concerns:

- Customer Concerns:
  - There is a perception that doing Agile is time intensive for customers. In a waterfall project, we, as customers, can dedicate our time only at certain slots around the major joint reviews, while in Agile processes we have to participate much more actively to a lot of meetings and invest much more time throughout the project.
  - In a waterfall process, decisions are taken at formal reviews with many reviewers, panels and boards. In an Agile process with frequent interactions between supplier and customer, many decisions are taken by the whole team at each iteration. How to avoid a change in the role and a shift in the ownership of risks and responsibility from supplier to customer?
- Supplier Concerns:
  - Agile Process can only be successful when there is a close collaboration with the customer. How do I get the required input and the feedback from the customer if they are not participating to the Agile process as they should?

Furthermore, the adoption of an Agile development method can introduce other challenges for the supplier and customer organisations, such as:

- The value of adopting an Agile process in a project is disseminated among all stakeholders.
- Possible additional effort due to new artefacts brought in by Agile methods (e.g. co-existence and maintenance of product backlog and requirements specification).
- How to align the system and sub-system development life cycles, particularly when one is not following an Agile process while the other is.

The rest of the handbook is intended to provide best practices and recommendations on how to address these concerns.

5 Guidelines for Agile life cycle selection

5.1 Selecting Agile

As Agile is a largely deployed software development method that has been pervasive across all types of businesses and organisations for many years, several suppliers have changed their usual software development practices to address various new challenges and Agile is one of the key enablers. Agile can bring in significant value over traditional practices; however experience indicates that it is not a silver bullet. Agile practices are developed based on several fundamental assumptions about the environment within which the project is executed. In the right context, all promises of Agile become realistic; but in the wrong context Agile can add significant risks to the successful delivery of the product. Please refer to clause 4.2 for Agile promises and risk related to its adoption.

This clause 5 aims at providing guidelines to:

- Help in deciding whether or not a project is suitable for Agile development based on criteria related to the environment and the product itself.
- Help in ensuring that the process that sustains the adoption is as well described.

5.2 Analysis of key factors for Agile selection

5.2.1 General

This clause provides a set of criteria or factors that play a role in the software development process, and that can help to guide the decision either for agile or against agile, but that can also help with the selection of the model in clause 6.

Concerning the Agile mind set, it is not very adaptive or agile to mandate one single approach with mandatory factors. A selection of factors that work best in the relevant environment is key for success. It is worth noting that theoretical fit and practical fit are often quite different.

There are some groups of people or organizations with interests in the result of the software development process, which in the most common case are the customer and the supplier as organisations. We also have to remember that these factors are just a support and they are not a replacement for thought and dialogue with the project stakeholders. Rather than using them in isolation, use them to start a conversation about Agile suitability and build consensus around the method of choice in close relationship with all the project stakeholders.

The process that handles the selection is the Agile Assessment. As explained on clause 5.3, the Agile Assessment is done as soon as possible, that is during the bid phase, and includes as a minimum the stakeholders.

There are also other aspects to take into consideration that are not project related. An organisation/company plays a critical role in assessing the suitability of adopting Agile and should therefore be part of this process. Even if the project is a great fit for Agile in theory, if management or other stakeholders are against that approach, then its application carries significant risk.

NOTE Non-project related factors, related to the organisation itself, are recommended to be included in the assessment.

These criteria can be grouped into 4 contexts as follows:

- Customer context: evaluation focuses on customer ability to receive frequent deliveries, to define the need and to collaborate with the project's team.
- Supplier context: evaluation focuses on the partner capabilities to develop the project and to deliver the product or the solution to the customer.
- Project context: evaluation focuses on the solution to be developed in terms of characteristics, architecture and design.
- Team context: evaluation focuses on team ability to perform Agile development (including size, co-location, multi-disciplinary)

Figure 5-1 shows the different factors/criteria that are covered by each context.

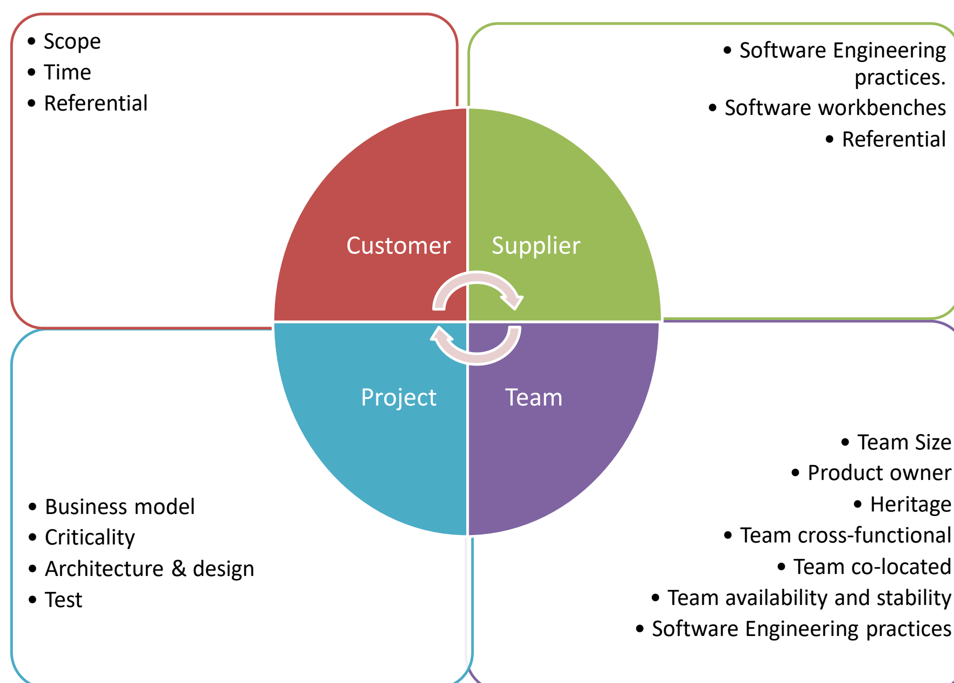

Figure 5-1: Factors for adopting Agile process

NOTE Not all the contexts and the underlying criteria are equally important and it is recommended to weigh them. This weighting is based on the experience of Agile experts and strongly related to the company environment.

All factors are evaluated according to the same scale with five values. This scale is depicted in Figure 5-2. For each context, an average is calculated using the scale.

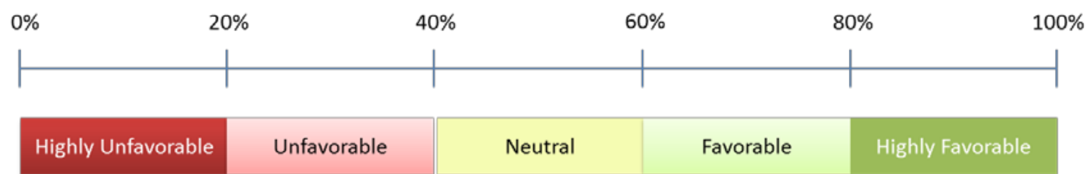

Figure 5-2: Agile selection factors scale

The global average can be calculated by applying weighting coefficients to the four individual context averages. The weighting coefficient would in this case express the perceived importance of a particular context and can be decided by the relevant stakeholders.

5.2.2 Customer context

The customer context covers the following factors:

- Scope: Are the requirements (or user-stories) of the project well known? Are the requirements stable throughout the development life cycle? Agile methods allow for change and uncertainty, then projects that have not well defined requirements are well suited to an Agile process. The refactoring effort involved for incremental development may not be justifiable when the requirements are frozen upfront. The Agile approach is more adapted when requirements changes requested after the beginning of the project are frequent and need to be taken into account quickly in a new iteration, but the agile approach can also be adequate when all requirements are well-defined and understood from the beginning, though some of its practices will not be needed.
- Time: Is the customer request for a rapid (i.e. first project release) and then regular incremental delivery? Agile methods bring value due to its emphasis on working software, therefore projects that require an incremental delivery with an early project release are well suited to an Agile process. Has the project a fixed time frame? The Agile approach is particularly useful for getting maximum business benefit within the given time. So Agile is favoured when intermediate software deliveries can provide added value to end user/customer.
- Experience and awareness: Does the customer understand Agile methods? Is the customer willing to change/adapt project management practices? This criterion allows to assess whether or not an Agile mind set is in the customer’s referential. Applying an Agile process in a project with a customer without any knowledge on Agile carries significant risks.

Context	Factor	Unfavourable	Favourable
Customer	Scope	Well-known and stable requirements	Not-well-defined or not-very stable requirements
	Time	No or few intermediate deliveries required	Intermediate deliveries required regularly.
	Experience and awareness	Lack of experience and mind set in agile.	Interest/experience in Agile

5.2.3 Supplier context

The supplier context covers the following factors:

- Software engineering practices: Is the supplier experienced with software engineering practices, which are typically used in Agile software development projects, such as continuous build and integration, test driven development and pair programming well mastered by the supplier? How long has the supplier been developing software using Agile process? How mature are the processes (and the people) relative to software development?
- Software workbenches: Does the supplier have the adequate software (e.g. continuous integration) workbenches in order to be able to deliver the project iteratively and incrementally? Does the supplier have the necessary knowledge and experience to use these workbenches?

Table 5-1: Supplier context

Context	Factor	Unfavourable	Favourable
Supplier	Software Engineering practices	Mature non-agile Software Engineering Processes	Mature Agile Software Engineering Processes
	Software workbenches	Tools and workbenches not available to support the agile process	Available tools and workbenches supporting agile, i.e. continuous build and integration engine and related tools

5.2.4 Project context

The essence of the Agile approach is to develop software iteratively and deliver it in usable increments that provide value to the business. The nature of the requirements, as well as the solution to meet them, play a key role in executing the project using Agile methods.

The project context covers the following factors:

- Business model: is the target an internal system, a commercial product, a bespoke system on business agreement for a customer, a component of a large system involving many different parties, an evolution of a large legacy system? This criterion allows assessing whether demonstration of direct business value for the customer using an agile lifecycle is easy or not. As an example, in software development from scratch, it could be favourable to adopt Agile because it aims at providing early working versions of the software from the very beginning. On the other hand, for developments based on legacy systems, it may not be easy to demonstrate additional business value through adoption of an agile lifecycle, if the migration does not add new functionality. This is partly because the business value of the application exists already with the legacy software. Also in such cases it may be difficult to achieve the required level of automation of the testing and continuous integration with the legacy applications.
- Criticality: What is the criticality of the project? Documentation needs to increase dramatically to satisfy external agencies who want to make sure the security of the investment and the safety of the public are guaranteed. Some critical activities may be governed by specific procedures or rules.
- Architecture and design: Is there an implicit de facto or reference architecture already in place at the start of the project? Does the project have well-defined external interfaces? Projects that have a reference architecture avoid a big up-front design phase, hence are well suited for an

Agile process. Projects that have external dependencies are not well suited to an Agile process. Those external dependencies can also include hardware constraints.

When developing software for a very constraining system (e.g. CPU, memory, real time constraints), it is preferable to start when the requirements are stable, and therefore be able to optimise it so Agile is not well suited.

- Test: Is testing an increment at sprint or release level too expensive in terms of effort, time and availability or test means? Is test automation easy to put in place? Is the test effort compatible with sprint or release standard duration? Typically if testing an increment is expensive, probably the team cannot afford to do it very often. Agile requests to demonstrate all the user stories within a sprint and check non regression as well. In a context, when test effort requires extending the sprint duration above 4 weeks, trying to apply an Agile method based on Scrum is not recommended. Projects that have costly testing environment (test harnesses) with regards to the project development itself are not well suited to Agile process. The Agile approach imposes the availability of integration and test means (real hardware or simulators or stubs) very early in the process to start with testing from the beginning on.

- Resource availability (e.g. hardware): Due to the anticipation/advancement and execution of all or some of the ECSS-E-ST-40 activities in parallel at early sprints of an agile life-cycle, additional need for availability of artefacts on which the software under development relies may be accordingly pushed forward in the schedule of the software development project. For instance the HW on which the software must executed, would be needed, if prototyping, coding and validation are already performed in early sprints. The same applies on the need for another software, which is required for integration and for validation purposes of the software under development. Conversely the agile life cycle can help to better specify the hardware specification through prototyping activities. Such considerations must be taken into account in the project context, when selecting an agile software development lifecycle. Possible project specific solutions/mitigations may be need accordingly (e.g. use of emulators, simulators, mock ups, etc.).

Table 5-2: Project context

Context	Factor	Unfavourable	Favourable
Project	Business Model	Legacy software, migration or evolution of an existing software system	Development from scratch or according to a reference architecture
	Criticality	Critical Software, e.g. flight software or numerical software verification facilities	Non-critical software, e.g. ground software
	Architecture and design	No existing reference architecture or not yet defined interfaces to other elements as software or hardware	Existing reference architecture or well-defined interfaces to other elements as software or hardware
	Testing	Difficult to automate tests. Complex testing environment.	Easy to define and automate tests. Simple testing environment.
	Resource availability	Dependency on artefacts external to the project (e.g. hardware)	Not clear dependencies on external artefacts (e.g. hardware capabilities to be decided based on prototypes)

5.2.5 Team context

Agile development favours individuals and interactions over processes and tools. The approach is more people-centric; hence, team composition plays an especially important role in assessing Agile suitability. The availability of the right skills and the team's ability to collaborate play significant roles in the success of the project.

The team context covers the following factors:

- Size: How many people are involved in the project team? The overall size of the system under development is an important criterion, as it drives the size of the team, the number of teams, the needs for communication and coordination between teams. Agile adoption recommends a small team, so teams with less than 8 people are well suited.
- Team cross-functional: Is the team cross-functional? Does the team have all skills to develop the project? Teams that contain all necessary expertise are in a better position to identify and resolve impediments as they occur. Having team members with different experiences and points of view provides more comprehensive feedback. Agile recommends that all team members are willing to learn and fosters a "generalist" approach.
- Team co-located: Linked sometimes to the size of the project. Is the development team co-located? How many teams are involved and are not collocated? This increases the need for more explicit communication and coordination of decisions, as well as more stable interfaces between teams, and between the software components that they are responsible for.
- Agile heritage: Does the project team have a previous experience in Agile process? This criterion allows to assess the Agile coaching effort to sustain the team and reduce the risk of Agile adoption.
- Team availability and stability: Is the team dedicated to project? How stable is the team throughout the development? Agile recommends stable and available team, so without this criterion deploying Agile in that context may increase risk.
- Product owner: Is there a single product owner who is fully dedicated to the project? Does the product owner provide end-user continuous feedback and commitment? A product owner available and working closely to the team in order to provide it with a continuous knowledge and feedback on user need is a key criterion for Agile adoption.
- Software engineering agile practices: Does the team have the relevant knowledge and experience in software engineering agile practices?

Table 5-3: Team context

Context	Factor	Unfavourable	Favourable
Team	Size	Big team > 9 people	Maximum 9 members
	Cross-functional	Highly specialized team	Specialized but cross-functional team
	Co-located	Geographically distributed team	Fully co-located team
	Agile Heritage	Experience in agile developments < 3 years for all members as an average	Experience in agile developments > 3 years for all members as an average
	Availability and stability	Team working on several projects.	Fully dedicated team to a single project.

Context	Factor	Unfavourable	Favourable
		Team changes over the project lifetime.	Team is stable over the project lifetime.
	Product Owner	Partial time product owner, or lacking domain knowledge or soft-skills	Fully integrated and available product-owner with soft-skills and good domain knowledge
	Software Engineering practices	Team members do not master agile practice	Team members master agile practices

5.2.6 Key Factors Summary

Table 5-4 lists the key factors considered to drive or prevent either the classical or agile lifecycles.

Table 5-4: Key factors for selection of classical or agile lifecycle

Lifecycle	Classical		Agile	
Factor	**Positive**	**Negative**	**Positive**	**Negative**
Customer: Scope		Unknown or unstable requirements		
Supplier: Software Engineering practices				Agile practices not well understood at company level
Supplier: Software Engineering Workbenches				Knowledge and deployment of tools not available at company level.
Project: Criticality	Critical software			Critical software
Project: Testing				Not easy testing, automation and environment.
Team: Size	Team size >9			Team size >9
Team: Heritage				No substantial agile experience
Team: Availability and stability				Team is shared among different projects.
Team: Product owner				No dedicated and knowledgeable product owner.

5.3 Agile assessment process

The Agile assessment process can be performed as early as possible so during the bid phase and/or development phase, the criteria are evaluated with the elements known at this given moment in the project. Then, the evaluation assessment is only valid for a given time period.

GOOD PRACTICE

It is recommended to perform the Agile assessment process as early as possible and re-assess it in a periodic manner.

Factors that can impact the decision within an organisation are provided in the four contexts described in clause 5.2. Furthermore, this evaluation is performed in a recursive way according to the Product Breakdown Structure (PBS) & Organisation Breakdown Structure (OBS) of the project under consideration.

GOOD PRACTICE

It is recommended to consider the global picture of the development. As a result of that, the Agile assessment process can be performed in a recursive way at each stage of the development organisation.

Finally, it is important that this engineering practice selects an appropriate Agile method and is led by someone with enough Agile experience and background from the organisation, acting as an independent assessor within the environment during a dedicated meeting with all the project stakeholders.

GOOD PRACTICE

After the project assessment, it is also recommended to have the commitment and the full support from management at the customer and supplier sides.

5.4 Selecting agile or waterfall

Agile is a method that can be applied in many cases and for many projects, but the traditional waterfall / incremental approach often can be a good choice as well. Table 5-5 introduces several aspects that can be reviewed in the selection process of the development approach for a project.

Table 5-5: Aspects for the selection of agile or waterfall approach

Category	Aspect	Lifecycle recommendations
End-user	Requirements stability	**Agile** approach is more adapted when requirements changes requested after the beginning of the project are frequent and need to be taken into account quickly in a new iteration. **Waterfall / incremental** also manages unstable requirements but its longer iteration cycle and more formalized process makes it inherently less reactive to changes.
	End-user commitment	Continuous feedback is important in the **agile** approach, but when direct customer/end-users involvement is not possible and the product owner is not able to represent customer/end-users needs, the use of **waterfall / incremental** approaches are preferred.
	Time to market	**Agile** and **incremental** are favoured when intermediate software deliveries provide added value to end user, or when these deliveries may have opportunities to provide concrete feedback on the software. These intermediate deliveries can only be useful when the end user has the ability to use and evaluate those intermediate deliveries. Functions currently not supported should not prevent from such evaluations.
Project management	Formalism	When high level of formalism at process / document level, or full specification/design/tests traceability demonstration is a core customer requirement, the **waterfall / incremental** approach can be easier to implement.
	Competence	**Agile** recommends that all team members are learning to perform all development activities and fosters a “generalist” approach. **Waterfall/incremental** cycles allows different types of team organizations (with either specialized team members or not), but usually for large projects/team sizes, team members are often assigned to one activity and don’t necessarily need to master other activities.

Category	Aspect	Lifecycle recommendations
	Risks	Through constant requirement prioritization, the **agile** approach can treat the risks earlier than **waterfall / incremental** approaches. Risk management is embedded in the process: Stand up meetings, reviews, retrospectives, demos, and the definition of done support it. **Waterfall / incremental** approaches work with a fixed set of requirements from the beginning of the project, the visibility on the overall technical risks is then higher.
Technical	CPU, memory, real time constraints	When developing software for a very constraining system, it is preferable to do the design when the requirements are stable, and therefore be able to optimize it. In this case, **waterfall / incremental** should be favoured or else **agile** methods be used very carefully, as natural requirement instability can lead to project costs overruns due to frequent and complex redesign.
	Test means	The **Agile** approach requires the availability of integration and test means (real hardware or simulators or stubs) very early in the process to start with testing from the beginning on. **Waterfall / incremental** approaches usually need integration platforms later in the project.
	Test automation	Test automation is always recommended whatever kind of cycle is chosen, and the higher the number of iterations required to be performed, the greater the benefits of automation. **Agile** being based on multiple short cycles including test, reaching a high level of automation is a condition if not a requirement for this method to be effective. A constraint such that only a low automation level is possible is less an issue for **waterfall / incremental**, as much less iterations are expected to be performed.

6 Reference models for Scrum-like Agile software life cycle

6.1 Introduction

The Reference Model presented in this clause 6 does not intend to be a comprehensive description of the Agile activities. It provides a reference software lifecycle inspired by the Scrum methodology and amended with specific activities, such as qualification and acceptance, which are typical and required in the context of software development projects for space missions. This formalism of activities, meetings and their correlation to joint reviews facilitates the mapping to the processes of ECSS-E-ST-40 and ECSS-Q-ST-80. This reference process model does not intend to be a handbook about Scrum, and does not aim to provide information on different ways to implement it and other technical aspects. Scrum was selected as a reference and inspiration, among the methodologies, because it is a commonly used approach in many Agile implementations and offers an open framework to integrate and combine with other Agile methods and techniques.

This clause 6 describes:

- Roles and competences.
- Overall Agile activities.
- Main meetings or key points.
- Organisation of Agile activities.
- Organisation of ECSS milestones.

6.2 Roles and competences

6.2.1 Overiew

Depending on the project complexity, criticality level, project size, scope, and other distinguishing factors, different stakeholders will be involved in one way or another. Typical roles found around the project often include: Domain consultants, SPA / QA engineers (as independent persons/roles), Project Manager, or even Business Analysts.

6.2.2 Scrum master

The Scrum master acts as a facilitator to the team. In particular, the scrum master is in charge of removing impediments that block the team, ensures that there is a proper communication between the team and the product owner, guides and coaches the team and protects the team from external

influences during the sprints. In addition to this, the Scrum master is responsible for the smooth running of the overall Agile lifecycle and organises the retrospective. The Scrum master therefore maintains some kind of neutrality inside the Scrum team. It is difficult to map to the role of Scrum master to an existing role in traditional software development.

Given these constraints, this role is ideally fulfilled by a third party, but this is difficult to fit into a traditional space project. Therefore, in many cases, the role of Scrum master is taken over by a senior developer, the software product quality assurance manager or the project manager. The Scrum master cannot be the same person that plays the role of product owner.

6.2.3 Product owner

The product owner:

- Defines the product by means of user stories
- Liaises with and represents the final users
- Interfaces the development team to discuss and clarify details of the product functionality
- Assigns priorities to the different user stories/functionalities of the product
- Takes responsibility for the outcome of the activity
- Conveys the vision and goals of the product at the beginning of every release and sprint

The product owner is as close as possible to the end customer and user and as knowledgable as possible to the business and solution context. That is, the product owner acts as the customer for the team. Depending on the level of implication of the parties in the Agile process, the role of product owner can be taken over by someone from the client side (often the project manager) or someone from the supplier side. It is also possible to have someone from the client side as formal product owner, delegating part of the responsibility on the supplier (this is often referred to as proxy product owner). The delegation of the product owner role to the client side becomes particularly relevant when the contract setup involves a customer, a supplier (prime) and a subcontractor of the supplier. A common scenario is when the prime develops the system and the subcontractor, the software to be run on this system. While it is clear that the prime can take the role of product owner and ensure that the backlog is updated in line with the requirements derived at system level, it is strongly recommended that the client interfaces regularly with the prime (or, even better, joins the agile meetings regularly), to ensure that requirements are clearly understood and that coordination exists between the three different levels.

6.2.4 Development team

The development team implements the product based on the inputs and guidelines from the product owner. It consists of a self-organizing and self-managed, cross-functional team of people who collectively are responsible for all of the work necessary to produce working, validated assets. It is a team composed of members with all the functional skills (such as UI designers, developers, testers) and specialties necessary to complete work that requires more than a single discipline. The team members knowledge is typically complementary.

6.2.5 SCRUM team

The Scrum Team consists of the Product Owner, the Development Team, and a Scrum Master. When the word "team" is used, it refers to the SCRUM team.

[SRUM]

6.2.6 Agile coach

In some projects, coaching can be considered as an option. An Agile coach, also called Agile referent, can offer more accurate and specific advice in order to support the Agile assessment, described in clause 5.3. An empowered Agile coach can be critical for the success of an Agile development project. The Agile coach has enough experience and background from the organisation to set up with the project stakeholders an efficient action plan to ease the Agile adoption.

Agile coaches help train corporate teams on the agile methodology and oversee the development of agile teams to ensure effective outcomes for the organization. They are responsible for guiding teams through the implementation process and are tasked with encouraging workers and leadership to embrace the agile method. The agile coach's ultimate goal is to arm agile teams with the right knowledge, tools and training so that they'll be able to use agile to its full potential.

6.2.7 Training and competencies

The following competencies are recommended for the different actors in order to run a project as Agile:

- General agile training (e.g. Scrum) is recommended for all actors
- Scrum master: Scrum master certification recommended. Previous experience in agile development is fundamental.
- Product owner: Product owner specific training.
- As a minimum either the product owner or someone in the development team should have been trained in Agile practices.

In addition, the whole organisation (at least all the stakeholders) needs to have basic Agile knowledge and awareness.

In many cases coaching can bring benefits (particularly if there is limited experience in the team). Coaching can be performed by a third party (outside the project) who has extensive experience in the execution of agile projects and good communication skills. An agile coach should typically have agile certification (e.g. Scrum master)

6.3 Exemplary Agile activities

6.3.1 Distinction between meeting or activity

Most Agile texts often refer to the planning meeting as opposed to the planning activity. For the purpose of these guidelines the following criteria were chosen:

- Activity: produces artefacts which are part of the development (output) based on inputs. Involves transformation work (converts inputs into outputs)
- Meeting: reviews artefacts produced as part of an activity, but does not transform inputs into outputs

This clause 6.3 proposes a number of activities with specific inputs, outputs and objectives for an Exemplary Agile methodology. The activities described hereafter have been inspired in Scrum common practice, taking also some Agile generic ideas in consideration. Most elements and the way they are used have been derived from [SCRUM] (e.g. “user stories”, “backlog”, “sprint review”, “retrospective”).

Figure 6-1 shows the activities to be performed during a typical iteration or sprint. It starts with the planning of the iteration itself. This planning can be divided in two parts: first to determine what will be done, which user stories or features will be implemented, considering the capacity available during the sprint, and second, how each user story can be initially decomposed into specific tasks for the team (to be noted that further decomposition takes place during the sprint). Ideally, the user stories are prepared beforehand (during the refinement meetings) so that no unnecessary time is spent in the planning meetings. During the sprint a number of activities are performed. Some of them are tied to the sprint (such as the coding itself) while others are of generic nature, i.e. not directly linked to the contents of the sprint. This includes the refinement and update of the product backlog and the collaboration between all implicated parties to obtain early feedback. The product owner improves the product backlog in collaboration with the stakeholders and the development team. Tracking of the sprint activities and management of the product backlog ensures that the plan for the sprint is followed or adapted if needed.

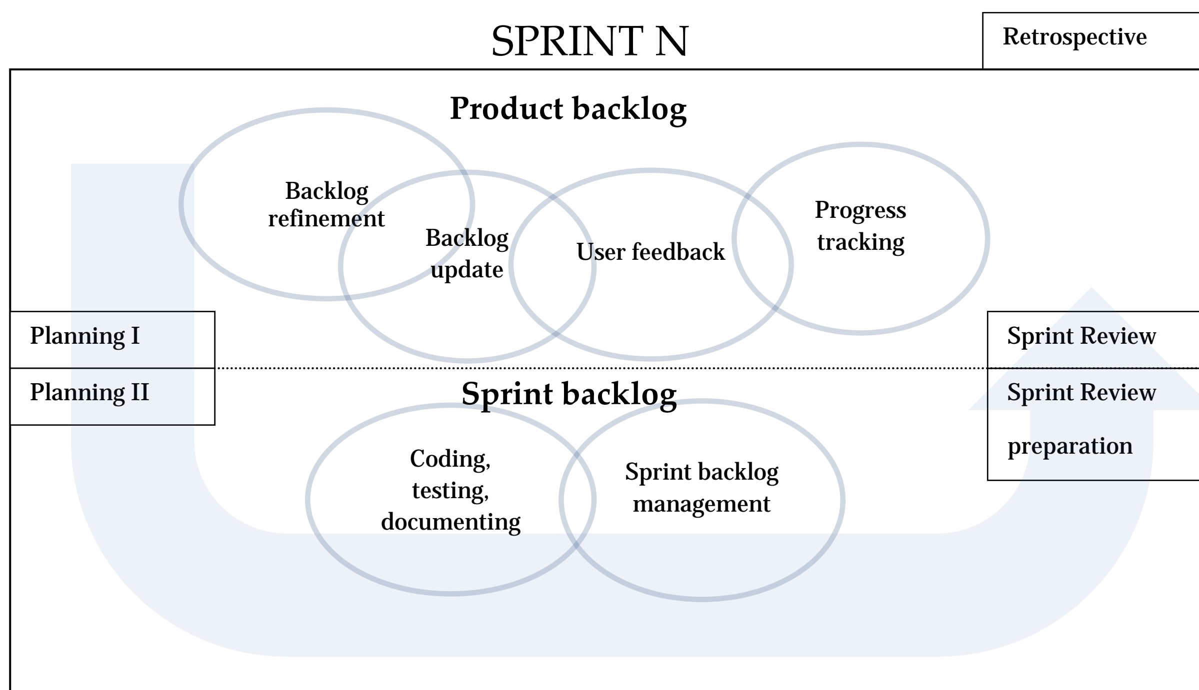

Figure 6-1: Organisation of activities during a sprint

The Sprint concludes with a review and approval by the product owner of the outputs of the Sprint, which typically include (but are not limited to) software and documentation, based on the Definition of Done (DoD). All stakeholders are invited to attend a demonstration of the implemented user stories. The sprint can be closed with a retrospective meeting, where good practices and improvements to the process are identified, often focusing on a few strategic changes or things to maintain for the next Sprint.

6.3.2 Planning I – What will be delivered

Input: Product backlog

Output: Sprint backlog

Objective: Define activities for a specific sprint (time boxed)

This first part of the Planning activity (equivalent to a first part of the sprint planning meeting in the Scrum methodology) shares many characteristics with the refinement and product backlog update activities. It is executed at least by the team, or at least by the product owner and a representative of the team (other team members can be appointed as necessary). During the activity, it is possible to create, refine and split user stories, as well as to allocate estimates. The key objective, however, is the selection of user stories for the next sprint. Although the activity is time-boxed it ends only when this allocation is complete. It is possible (and recommended) to leave additional user stories (e.g. 20% of sprint time) ready for execution (with clear priorities) in case the allocated ones are completed or are abandoned due to unforeseen circumstances, such as new priorities.

It is advisable to start by defining the duration of the next sprint and then set dates for the next planning and review meetings. The sprint typically lasts between 2 and 4 weeks, although less than four weeks can often be inefficient if a full set of documentation needs to be updated as part of the sprint. In some specific circumstances it is possible (although typically not advisable) to do a shorter or larger sprint. Short sprints can be used when the requirements are very unstable or just before review activities (while the documentation is reviewed by the stakeholders). The capacity of the team can be estimated at this point and then the selection of user stories can start. The team capacity can be estimated in days, depending on how the product backlog estimation is done.

The allocation of user stories takes into account the following points:

a. priority of user stories (e.g. expected business value or knowledge value for the team)

b. dependencies between user stories (although the user stories are independent from each other, in practice this is not always easy to achieve)

c. Specific capabilities and maximum time allocation of teams involved

Point 6.3.2c can be relevant for projects that are executed by more than one team, where each team has its own area of expertise and even their own allocated maximum number of hours. While not a standard Agile practice, it is advisable to take into account this factor. In fact, it can be considered an advantage in the sense that it allows the product owner to make the best of the team’s expertise.

6.3.3 Planning II – How will it be delivered

Input: Sprint backlog

Output: Updated sprint backlog

Objective: split the activities of the Sprint backlog into tasks, agree on how to implement the items and do the initial split of the tasks among the developers.

The second part of the Planning (equivalent to a second part of the sprint planning meeting in SCRUM) is conducted internally by the team, although the product owner is available for clarifications (and also can attend on request). This activity takes the sprint backlog as input and converts user stories into tasks. An initial allocation of tasks is done at this point. This part of the planning involves the whole team and produces the first version of the sprint backlog. It ends when all user stories have been converted into tasks. If this is not possible due to time constraints or other factors it should achieve at least the following:

a. there are enough tasks allocated for the first 3 days of the sprint,

b. the team can establish that there is no overcommit and

c. all architectural implications are clear.

6.3.4 Sprint backlog management

Input: Sprint backlog

Output: Updated sprint backlog

Objective: Select activities for immediate execution and update the sprint backlog status

This activity can be executed by any member of the team. It is therefore crucial that the sprint backlog can be accessed and maintained in parallel by all members (the simplest example is a board located in a central place where all members have access).

The activity consists of development team members selecting from sprint backlog which tasks they want to implement. This can in some cases involve also defining new tasks, and updating the tasks status.

6.3.5 Product backlog refinement

Input: product backlog, user input

Output: updated product backlog

Objective: upgrade the product backlog in terms of priorities, estimate the effort for each product backlog entry

This activity is quite central to any Agile development. It starts with an existing product backlog, which can be empty, and ends with an updated version, which, in most cases, is not final.

The activity involves the product owner, at least one representative of the team and, in some cases, the final users. Other members of the team and stakeholders participate as needed. If applicable, the meeting can be supported and coordinated by a Scrum master.

During refinement:

- User stories are added (by any of the participants).
- Acceptance conditions are defined.
- User stories are prioritized (e.g. expected business value or knowledge value for the team).
- Estimations are allocated to user stories.

NOTE Estimates are usually in points or days.

Newly added user stories are marked as “created” and transitioned to “ready” as soon as a priority and an estimate is agreed. Different approaches can be used to agree on the estimate (see 7.2.2.5). It is,

however, recommended to have at least the agreement of the team representative. User stories already "in progress" are not expected to be modified (unless this is done for clarity purposes), although there are cases where user stories need to be stopped during the sprint (e.g. due to unexpected implications found during the implementation such as impact on other user stories). Similarly, user stories "done" cannot be changed (although a new user story can always be created).

6.3.6 Progress tracking

Input: real effort as recorded, product and process measurements

Output: progress report and performance report

Objective: Produce a report of the effort spent for a fixed period of time and a report for performance indicators

This activity is done by the Scrum master (or a senior developer) and reflects the real effort spent so far by the team. Progress tracking and performance indicators are defined for each specific project. Progress report, in most cases, measures the difference between the real effort spent and the effort originally allocated to the activities. Performance indicators characterize adhesion to the agile process (e.g. number of rejected user stories, user stories volatility, work in progress measurement, cycle time, etc.).

6.3.7 Product backlog update

Input: product backlog

Output: updated product backlog

Objective: add user stories to the product backlog and organise them by priority

This activity is run by the product owner with the support of the product users and consists in defining new user stories and updating existing ones. Priorities, in particular, are often revised and refined during the activity. They can be changed as deemed necessary as far as the relevant user story is not "in progress" and is not "done" yet (bearing in mind the considerations already exposed in the previous clauses).

6.3.8 Coding, testing and documenting

Input: code, test plans and documentation

Output: updated code, test reports and documentation

Objective: fulfil and close an open task by implementing the relevant functionality, validating it and documenting it

This activity can be executed by any team member or by several team members simultaneously (for example pair programming sessions). The objective is to create the specific code to fulfil the objectives of a particular task. This code needs to be validated and documented properly. Test driven methodologies are often used during this activity, i.e. the tests are created and in some cases even executed before the code is written. Although in case of test execution, the test failure may look like an obvious outcome, it provides some level of reassurance in the sense that badly written tests may not fail if they are targeting the wrong functionality or it may help in identifying issues such a non-working test reporting system.

6.3.9 User feedback

Input: product backlog, documentation, code

Output: updated product backlog

Objective: Confirm that users of the product agree with the work done so far and gather information for the next sprints

Although accepted Agile practice recommends to involve the users as needed in any activities that have an impact on the product backlog (i.e. first part of the planning, review and refinement), this is not always possible. Therefore it is quite important to discuss the progress so far with the users. In order to minimize the impact of the feedback, it is recommended to regularly discuss with the users the status of the product. This is done at least once per sprint so that the users can see the outcome of the previous sprint. The feedback from the users do not impact user stories which are in status "in progress" or "done" (i.e. approved by the product owner during a review meeting). If this happens, a new user story is created.

6.3.10 Review preparation

Input: Sprint backlog

Output: Demonstration

Objective: Consolidate test plans to prepare a demonstration that shows that the user stories have been implemented

It is executed by the team members and consists in preparing the review activities, where the team demonstrates that the user stories have been correctly implemented. Therefore, it is critical to have a suitable script to run through the demonstration.

6.3.11 Sprint review

Input: product backlog, documentation, code and test plans

Output: product backlog

Objective: Confirm that the artefacts created during the sprint are aligned with the user stories and validate the implementation of the latter.

During the review (equivalent to the sprint review meeting in SCRUM), a demonstration of the product takes place and the test plans are eventually executed to prove compliance with the user stories allocated to this sprint. As soon as a user story is demonstrated and accepted by the product owner, its status in the product backlog can be moved to "closed". The user stories are demonstrated based on:

a. the Acceptance criteria, and

b. the Definition of Done.

The acceptance criteria describe unambiguously what needs to be demonstrated in order to consider a user story completed and accept it.

The DoD is applicable to every user story and spells out additional criteria needed to consider a user story complete. It is defined and agreed early in the project and can be modified at any time, although it cannot be applied retroactively, i.e. user stories in progress or already accepted are not impacted by the changes. The DoD can contain, among other criteria, the list of documents to be updated, the

applicable coding standards, the relevant quality models to be applied and the architectural constraints.

It is worth noting that the successful execution of tests (possibly including regression testing) has to be included in the acceptance criteria or the DoD in order to be used for the acceptance.

During the sprint review new user stories are often created. They correspond to user needs that can only be identified after certain functionality appears and user needs that were not correctly spelled out at the time of writing the user story. If there are some non-compliances with the original user story, these non-compliances can be added as software problem reports to the product backlog, and are written in a much simpler way than user stories. In the case of major non-compliances the user story is not accepted (i.e. it is not closed) and then can be considered for moving it to the next sprint or set back to the “ready” (or even just “defined”) state.

6.4 Meetings

6.4.1 Daily meeting

Review targets: team feedback

Reference: Sprint backlog

Actors: Scrum master, development team

The daily meeting is a very short meeting, time boxed (15 minutes) and the objective is to present the work that will be performed during the day and to identify what problems need to be solved that impede progress.

6.4.2 Management meeting

Review targets: progress report

References: product

Actors: product owner and Scrum master (in some cases a subset of the team or the whole team can also be present)

The objective of the management meeting is to review the effort spent up to date and confirm that it is in-line with the results achieved so far. The meeting normally takes place every one or two months and is used to ensure that the project is on track.

6.4.3 Retrospective

Review targets: performance indicators, feedback collected

Reference: estimates for user stories

Actors: Scrum master (moderates), development team, product owner

The objective of the retrospective meeting is to review the performance indicators as well as other non-formalized inputs and agree on potential improvements to the different activities. In addition to this, the meeting identifies critical problems that need to be addressed as part of the management effort. This meeting is often split in two parts: one to verify how the development is progressing and one to verify the process itself (i.e. to check how the methodology is being applied and to identify potential improvements).

As part of the retrospective it is discussed what went well during the sprint and what could be improved (making continuous improvement a reality). The Scrum Master, supported by the team, should identify a number of practices (for example three) to keep doing and the same number of areas to improve during the next sprint.

6.5 Organising the Agile activities and meetings in a project to create a life-cycle compliant to ECSS-E-ST-E-40

6.5.1 Preliminaries

Before the start of the activities, the draft product backlog is prepared and made available by the product owner. Such draft product backlog has a set of user stories prioritised. This is achieved by the product owner by running the product backlog update activities. The product backlog at this stage is made up of user stories and can be derived from the tender documentation, from existing requirements or from previous activities and collaborations between stakeholders. In some case, however, the product backlog can be the starting point for defining requirements. In most projects, user stories can be prepared for the creation and formalisation of test plans, code coverage measurements and quality analysis. Even if such activities are foreseen in the Definition of Done, the results need consolidation at some stage.

Activities such as the creation and formalisation of test plans, code coverage measurements and quality analysis are usually prepared in technical stories and not user stories. They have a specific DoD and acceptance criteria and unlike user stories, they are not described in a user oriented formalism.

Hence the recommendation is to define specific user stories for this, or technical stories.

There are some differences between a set of user stories and a set of traditional requirements. User stories are complete in the sense that they define a concrete purpose-built functionality that can be implemented and tested on its own (end-to-end feature). Furthermore, the acceptance criteria of the user story are part of this definition and contain all the relevant performance and documentation constraints. Traditional requirements, on the other hand, describe specific aspects and functions of the system. They often contain generic statements and describe the system successfully only when seen as a whole.

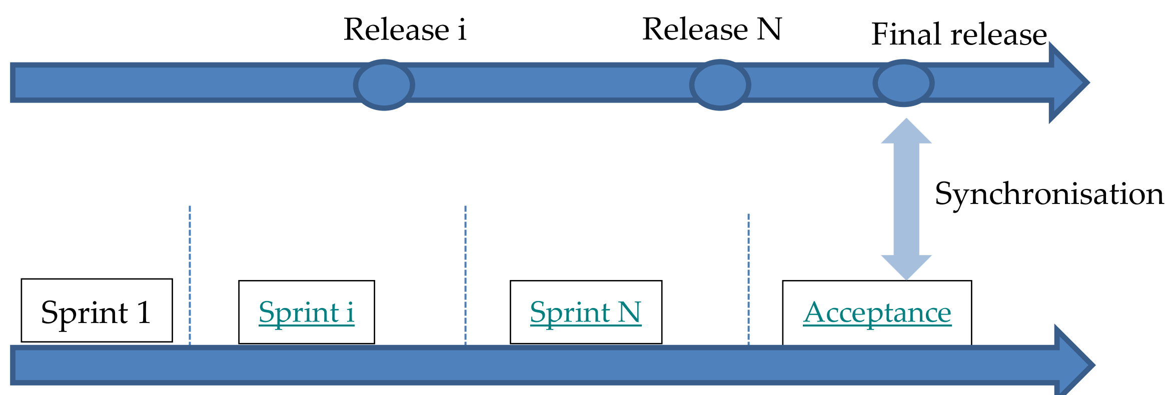

Figure 6-2: Exemplar Agile lifecycle

6.5.2 Product releases

6.5.2.1 General

Each Agile sprint may include activities that implement and validate specific functionality (the user stories of the sprint backlog). During the review meeting this functionality is demonstrated based on the acceptance criteria for the related user stories and the definition of done. The output of the sprint, however, is not necessarily a product release. Product releases are provided at specific milestones (e.g. every 5th sprint) and typically organised by grouping of user stories and functionality or components of the software. A product release is associated with the successful execution of acceptance activities (including the related QR/AR).

Product releases are not necessarily associated to or aligned with the end of sprints. It is theoretically possible, yet impractical, to end every single sprint with a product release. The Acceptance activities are executed on product releases not on intermediate working software (produced at each and every sprint). Since qualification and acceptance reviews are always associated to the execution of test activities, it is possible to conclude that the acceptance of a product is driven by the test plans associated to the product.

6.5.2.2 Qualification of a software product

A product is considered qualified, when a successful qualification review has taken place. The qualification review can be integrated in the Agile lifecycle or executed as a separate activity. Whenever possible, it is recommended to run the qualification activities, or at least part of them, while sprints are still planned, so that potential shortcomings can be addressed as part of the next sprint planning.

6.5.2.3 Acceptance of a software product

A product is considered accepted when an acceptance review has taken place. As with qualification reviews, acceptance can be integrated within the Agile lifecycle or run as a separate activity. It is recommended to end the development with an acceptance phase where formal acceptance of all the work produced so far takes place.

6.5.3 Start of the project: Sprint#0

The concept of "Sprint#0" is not part of the methodology described in standard Scrum, but it can be helpful when running the project within a particular environment, for example when it builds on existing software or reuses a set of tools. It comprises the first few activities executed right after the start of the project. It is intended for setting up the system and clarifying the user stories.

During this early sprint the team spends most time in setting up the development environment. This includes preparing the tools that are used during the project, getting familiar with the code or documentation, installing any customer furnished items provided as a starting point and installing the machines that are used for development activities. Sprint#0 typically lasts about 2 weeks.

In addition to providing the initial environment, Sprint#0 provides the opportunity to run the product backlog refinement activities at least twice. This is often enough to create a reasonable number of "ready" user stories that can be allocated as soon as the next proper sprint starts.

6.5.4 Development phase: Sprints #1 - #N

Sprints have variable duration, typically between 2 and 4 weeks. In most cases, 4 weeks is the most reasonable option as shorter duration can increase the administrative burden (additional meetings within the same time period) while longer durations introduce risks (since development can go into the wrong direction). In order to reduce the learning cycle and the tunnel effect, a short duration is preferred. In most cases four weeks is the most reasonable option for teams not fully experienced with agile process.

Each sprint comprises one instance of the following activities:

- Planning – What?: this allocates user stories for the current sprint and clarify priorities. The number of allocated user stories is sufficient to let the developers work during the relevant sprint. This activity marks the start of the sprint
- Planning – How?: this allows the sprint backlog to be created and formalised. It follows after Planning - What?
- Review preparation: Creates the artefacts needed to support the review activities at the end of the sprint.

During the sprint, the following activities are run an undetermined numbers of times in no specific order:

- Product backlog refinement: this estimates and prepares additional user stories for the forthcoming sprints
- Product backlog updates: this creates new user stories to be added to future sprints
- Sprint backlog management: this serves to create and allocate tasks to fulfil user stories
- Progress tracking: this creates the documentation necessary to support management meetings
- Coding, testing and documenting: this is the key process that ultimately creates the product itself.
- User feedback: this is crucial to gather information for future activities and to confirm that the work done so far is correct. Most user feedback, however, is collected during the review meetings, at the end of the sprint.

6.5.5 Acceptance phase

This phase takes place outside the Agile sprints, as most of the activities are performed by the customer/user. It is dedicated to the acceptance (and possibly qualification) activities and can only take place when a candidate product release is available, when enough functionality is implemented and the relevant documentation, including test plans, exists. It is possible to plan two or more separate acceptance phases for a given project. The acceptance includes formal delivery from the supplier as well as the installation of the product, eventually supported by the supplier.

6.6 Software lifecycle definition

6.6.1 ECSS-E-ST-40 reviews

ECSS-E-ST-40 considers the following main reviews:

- System Requirements Review (SRR): "This establishes the functional and the performance requirements baseline (including the interface requirement specification) (RB) of the software development" (ECSS-E-ST-40 clause 4.2.2)
- [optional]Software Requirements Review (SWRR): "In case the software requirements are baselined before the start of the architectural design, the part of the PDR addressing the software requirements specification and the interfaces specification shall be held in a separate joint review anticipating the PDR, " (ECSS-E-ST-40 clause 5.3.4.2)
- Preliminary Design Review (PDR): "To review compliance of the technical specification (TS) with the requirements baseline, to review the software architecture and interfaces, to review the development, verification and validation plans." (ECSS-E-ST-40 clause 5.3.4.2)
- Detailed Design Review (DDR): "In case the software detailed design is baselined before the start of the coding, the part of the CDR addressing the software detailed design, the interfaces design and the software budget shall be held in a separate joint review anticipating the CDR" (ECSS-E-ST-40 clause 5.3.4.3)
- Critical Design Review (CDR): "To review the design definition file, including software architectural design, detailed design, code and user's manual" (ECSS-E-ST-40 clause 5.3.4.3)
- Qualification review (QR): "This process is intended to confirm that the technical specification and the requirements baseline functions and performances are correctly and completely implemented in the final product." (ECSS_E-ST-40 clause 4.2.6. [on the objectives of the software validation process, which ends with QR])
- Acceptance review (AR): "The acceptance review is a formal event in which the software product is evaluated in its operational environment" (ECSS-E-ST-40 clause 4.2.7)

Depending on the objectives and characteristics of the project, the review objectives varies.

Both ECSS-E-ST-40 and ECSS-M-ST-10 state that the project is to be split into different phases and that reviews shall mark the transition between phases. ECSS-E-ST-40, however, leaves some flexibility to decide the level of completion of each phase and each review. More specifically 5.3.2.1 states

> *"5.3.2.1*
>
> a. *The software supplier shall define and follow a software life cycle including phases, their inputs and outputs, and joint reviews, in accordance with the overall project constraints and with ECSS-M-ST-10."*
>
> b. *The life cycle shall be chosen, assessing the specifics of the project technical approaches and the relevant project risks.*
>
> c. *The software supplier shall define the development strategy, the software engineering standards and techniques, the software development and the software testing environment.*
>
> d. *"The output of each phase and their status of completion, submitted as input to joint reviews, shall be specified in the software life cycle definition, including documents in complete or outline versions, and the results of verification of the outputs of the phase."*

A major implication of these Sub-clauses is that the activities of a phase do not have to be fully completed to move to the next one. Based on these premises, this handbook proposes four different approaches to implement a lifecycle based on the proposed reference framework.

ECSS-E-ST-40 clause 5.3.6 imposes some constraints on the way software level reviews take place compared to system reviews. For example, 5.3.6.1-b-2 requires for flight software that the software PDR takes place between the system PDR and the system CDR. This somehow implies that system level constraints will have an impact on the selected lifecycle. While this impact cannot be evaluated without knowing the specific project constraints, the next clauses will try to take it into account when presenting the different approaches.

General considerations about ECSS-E-ST-40 review objectives in the context of an agile software development lifecycle:

- SRR and SWRR can serve to obtain a common understanding of the initial product backlog, limiting the scope of work but keeping certain flexibility and to agree on further requirements elaboration and priorities with focus on interfaces.
- PDR Can serve to focus on the flexibility and scalability of the architectural design, as well as to establish a more constrained product backlog where critical user stories have been fully developed. A general validation approach to the software system can also be discussed.
- CDR can serve to establish a clear view of which requirements are implemented. Acceptance procedures can be established at this point.
- QR/AR. The objectives do not differ much from the regular QR/AR. In an agile software development lifecycle these reviews are typically applied at Product Release level. The purpose is to check the full extent of the qualification and obtain customer acceptance of the software system.

6.6.2 Organising the ECSS-E-ST-40 reviews in an Agile software approach

6.6.2.1 General

There are several options to introduce project reviews in Agile software developments. Formal milestones enable the progress of the project to be controlled with respect to cost, schedule and technical objectives from a high level.

Each sprint per se can be considered as a milestone, since there is a full review of the scope of the release, by examining requirements, design, tests and verification activities. However, having the involvement of decision-makers and a team of reviewers in each sprint review is unrealistic. The interaction between system and project review, complicates this further.

In the Agile approach the software product is constructed in iterations, and the final requirements specifications, design and tests can only be presented when the product is released. Agile methodologies value quick responses to change. This implies, in practice, that requirements, architecture and code can be changed at any point during the lifecycle if it is necessary and justified. It is important to state that the Agile process does not “promote” the change for the sake of change but attempts to better manage the necessary and unavoidable change. The source of change in the software requirement, design and implementation is independent from the adopted software process. The agile methodology enables the detection of the source of change earlier in the software development lifecycle. For instance, through closer involvement of users and through frequent release of software, the possibility of detecting software defects throughout the software development process is higher than compared to a waterfall process, when such defects would only be detected during the

validation phase towards the end of the life-cycle. The same holds true for the need for adjustment of requirements through feedback from the users. This is to say that regardless of whether an Agile process or a waterfall process is adopted, the need for the change is outside the process. The assessment of the necessity and justification for a change is also independent form the selected software life-cycle process. In practical terms, in a water-fall process, the mechanism adopted to handle the change are delta reviews and maintenance processes. The contractual and financial impact of the handling of changes in a water-fall process and in an agile process are not governed by the ECSS-E-ST-40, hence also outside the scope of this handbook. The different models proposed try to combine this flexibility with the constraints imposed by higher level systems and project requirements.

The models are summarised in Table 6-1. They are classified based on five associated potential risks, which have been categorised as “high”, “medium” and “low”. The association of any of those risks to a specific model means that the likelihood of such risk is considered objectively higher compared to other models under the same circumstances, but its final likelihood will depend on other factors:

- Inadequacy of final product: the associated risk is to produce deliverables that do not fulfil the required functionality. This measures how difficult is to identify and react to changes required to achieve the functionality. Changes are easily identified when working deliveries are produced at regular intervals (it should be noted that this does not refer to a formal software release). Quick reaction to change occurs when such changes can take place often.
- Additional effort required: the associated risk is to exceed the effort originally foreseen. In general, unforeseen work can be minimised by a) ensuring that the implemented software fits the required functionality and b) avoiding the repetition of activities. In the context of agile, the use of quick prototyping techniques and the delivery of working software on every sprint ensures that it fits its intended functionality. On the other hand, if the software is subject to formal approval, there may be a need to repeat activities.
- Increase in formal reviews: the risk discussed here deals with unforeseen increases in the number of formal reviews. A high risk indicates that formal reviews which include all the relevant stakeholders who are required to approve parts of the software can easily grow. Some models introduce the concept of implicit reviews, where stakeholders delegate to the product owner their responsibility (although product owners can request the participation of additional people at their discretion). Implicit reviews are not counted as formal reviews.
- Lack of endorsement by stakeholders: this refers to the formal acknowledgment and approval by the different stakeholders of the implemented software. The risk is potentially high when the formal involvement of the stakeholders during development is limited. This also applies when the stakeholders are invited to the different meetings, but there is no formal evidence of their agreement.
- Difficulties in coordination with system activities: this refers to how easy is the coordination between the system and software levels or system and sub-system software levels. Since this coordination takes place during meetings, it is easier to achieve if the number of formal meetings is reduced. A large number of formal meetings would make the coordination more complex. A complete lack of formal meetings would make this coordination extremely difficult. Another aspect relates to the synchronisations required between the system and sub-system reviews regarding the inputs from sub-system to the system reviews.

Table 6-1 – Overview of the different models

	inadequacy of final product	Additional effort required	Increase in formal reviews (and resources)	Lack of endorsement by stakeholders	Difficulties in coordination with system activtites
model 1	High	High	Low	Low	Low
model 2	medium	Medium	Low	Medium	Low
model 3	Low	Medium	High	Low	medium
model 4	Low	Low	Low	Medium	medium

In a waterfall software development lifecycle, the processes of the software development are executed (at least in theory) in sequential order, i.e. first requirements engineering, then design, then implementation, then validation …

In an agile software development lifecycle the processes are anticipated and executed in parallel. In practice this means that in early sprints (it can be even in the very first sprint) some or all of the software development processes are executed in parallel for the scope of the user stories of that particular sprint.

This brings the main challenge of how to deal with reviews. The logic of the ECSS-E-ST-40 reviews follows the sequential logic of processes. When mapping the ECSS-E-ST-40 reviews to the Agile life-cycle processes, depicted above, two main concerns need be addressed:

- How to avoid wasting effort: i.e. if something has been done in previous sprints, how to avoid that it is questioned and must be undone during a joint-review at a later point in time
- How to incorporate change vs baselining and freezing software development artefacts

The four models below try to answer these two points by taking the project needs and constraints into account. Some of the models make reference to "implicit" reviews. This refers to reviews or activities described in 6.5 that are part of the standard agile life-cycle and complement in content and activities one or more joint reviews. When these activities/reviews are run at regular intervals, they can anticipate some of the activities typically performed at joint reviews. Each model aligns best towards some significant constraint or characteristics of the project.

6.6.2.2 Model 1: Review driven lifecycle

Tag line: Agile hidden under the Waterfall.

The main characteristic of this model is that the review schedule has to account for external needs (e.g. spacecraft level, system level, system design, customer process, etc.).

Objective

The main characteristic of this model is that certain artefacts need to be available at certain points in time defined by the reviews. This typically implies that all the outputs of each development phase before the review takes place. This means that all intended requirements will be completed by the time SRR/SWRR take place and the architectural design must be ready when the PDR takes place and accordingly the coding shall be completed and validated against the Requirements by the CDR.

It is understood, that some changes will be required as a consequence of the review, but no changes will be allowed after this has taken place (outside standard formal mechanism). Although in practice this means that changes should be limited as much as possible and always with a good justification that will formalise the change.

This is a typical scenario when the system is developed following a waterfall lifecycle but the subsystems are developed following an agile approach.

Advantages:

— Limited numbers of reviews,

— Reviews can be agreed in advance and it is easy to involve all stakeholders. As a consequence, coordination with system level is easy

— The output of the development project is formally agreed by all stakeholders

Disadvantages

— Limited flexibility if many changes become necessary after the joint reviews.

— As producing working software is not the priority during the early sprints, some errors (e.g. wrong architecture) may remain undetected for several sprints and may increase the amount of additional work needed after the development. This model, inherits to a large extent the disadvantages of a waterfall life-cycle, since the sequential execution of activities and processes is still maintained.

Description

The reviews are fixed in advance (possibly based on the external constraints to the project). Work during the sprint aims at producing the outputs required at each review (although this does not forbid the use of agile practices during the early sprints and anticipation of other ECSS-E-ST-40 processes in parallel such as architectural design, coding and validation. The focus and the main driver for this model remains however the need for having the input for the joint reviews ready at pre-defined given milestones. All work done in parallel in the context of the agile software development life-cycle, e.g. the development of prototypes, remains at the supplier’s risk, since the endorsement is only performed in the context of joint reviews and not during sprint reviews. Hence, resulting changes must be expected and absorbed by the supplier). The main artefacts (e.g. requirements, architecture) are frozen after the relevant reviews. It is worth mentioning that this model has been for many years typical in software developments, where as seen from the customer perspective a waterfall process is followed, while the supplier often anticipates the work (e.g. assuming a design and starting coding early on in the project). The important aspect of this model is the formalism of endorsement and the related assignment of responsibility for the changes introduced after and before the endorsement at a joint review.

Suitability

This model is recommended when the requirements, design and implementation need to be agreed and/or coordinated in advance (e.g. at system level). Another good match for this model is when requirements and the expected design are well-known right from the beginning of the project (e.g. in case of high level of reuse), or there is ample expertise in developing the same type of software. Another scenario where this model suits well is when the software development lifecycle needs to be synchronised with the system development lifecycle.

Compared to a waterfall lifecycle, this approach splits the creation of artefacts, into several sprints (eventually supporting them with prototyping activities), ensuring that the official review goes smoother. The sprints leading to a joint-review can be anticipations of that review. This allows achieving a high level of maturity before conducting a join-review.

Variations

Variation 1: It is possible to do a waterfall lifecycle until the architecture is fixed, followed by an agile development phase.

Variation 2: A potential alternative is to run the SWRR and SRR reviews after one of the first product releases is available. Afterwards the requirements baseline would be frozen, and an architecture would be prototyped, and the scope of the project should be almost fully defined. After a subsequent release, following a set of sprints, the architecture and high level design should be fixed. Therefore the remaining sprints would not modify requirements and would just implement the pending part of the backlog to be implemented.

Figure 6-3 shows as example a first phase that focuses primarily on the requirements with a secondary focus on architectural design and some coding/prototyping until sprint 6 where the Software Requirement Review takes place. Then it focuses on the design until sprint 10 with Preliminary Design Review, but at the same time, coding becomes the secondary priority. And finally the focus shifts completely to the detailed design and coding.

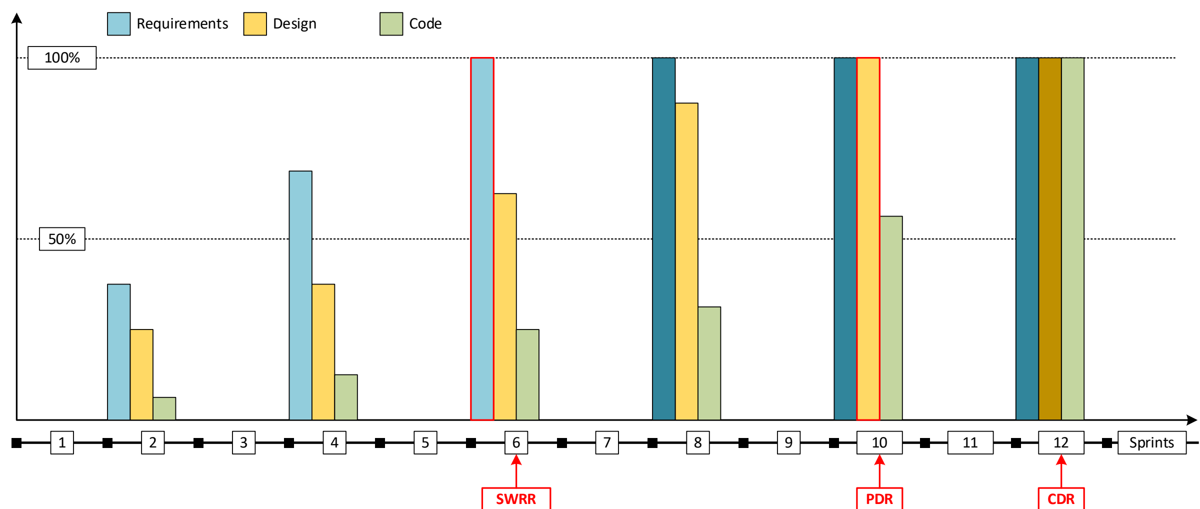

Figure 6-3: Model 1: Review driven lifecycle

6.6.2.3 Model 2: Review driven lifecycle with some flexibility

Tag line: Target of a high % of inputs to the review are frozen.

The main characteristic of this model is that the review schedule has to account for external needs (e.g. system design, customer process, etc.). The difference to model 1 is that the scope of the formal review applies only to certain parts of the software system under development (e.g. definition of interfaces, decomposition of the architecture).

Objective

Complete the outputs of each development phase before the formal review takes place up to a certain degree. This means that a certain percentage of requirements will be completed by the time SWRR takes place and the architectural design will be "almost" ready when the PDR takes place. Limited changes are still allowed after each formal review. This is a typical scenario when parts of the system need to be agreed upfront (e.g. interfaces or high-level architecture), but there is some degree of flexibility in the implementation. This could be the case, for example, if the system development is to be split among different suppliers.

Advantages:

— Limited numbers of formal review

— Reviews can be agreed in advance and it is easy to involve all stakeholders, making it easier the coordination with system development

— Some flexibility is possible

Neutral

- While most decisions are agreed at the reviews and therefore endorsed, there are activities that are not fully agreed
- Flexibility exists but is somehow limited
- Partially working software produced at each sprint ensures that the requirements and design are, at the very least, partially tested. On the other hand, most requirements and the overall architectural design is subject to approval, which could result in the need to redo the work

Description

The reviews are fixed in advance (possibly based on the external constraints to the project). Work during the sprint aims at producing a certain percentage of the outputs required at each review (although this does not forbid the use of agile practices during the sprints and anticipating activities from later phases to the earlier sprints). The main artefacts (e.g. requirements, architecture) are frozen after the relevant reviews to the extent that they have been produced.

Additional requirements can be added to the system but changes to approved requirements are no longer possible. The level of formalisation is quite high but some additions can be introduced into the project. After the initial formal review, the follow up sprint reviews implicitly comply with the joint review objectives in ECSS-E-ST-40. This approach requires an agreement regarding the extend of output completion that need to be fixed for the review. This can be agreed in advance and can be based on different criteria, including: areas or domains that need to be fixed, amount of resources implicitly or explicitly committed, work packages to be addressed.

Difference to previous model

As in model 1, this model produces highly formalised outputs that can be agreed by all parties and the outcomes are predictable. The main difference is that this model allows a limited amount of additional functionality to be introduced (but no changes to already approved requirements)

Suitability

Recommended when the requirement, design, implementation of (parts of) the system need to be agreed in advance but there is some scope for flexibility

Variations

It is possible to do a waterfall lifecycle until most of the architecture is fixed, followed by an agile development phase, which can include the introduction of additional requirement

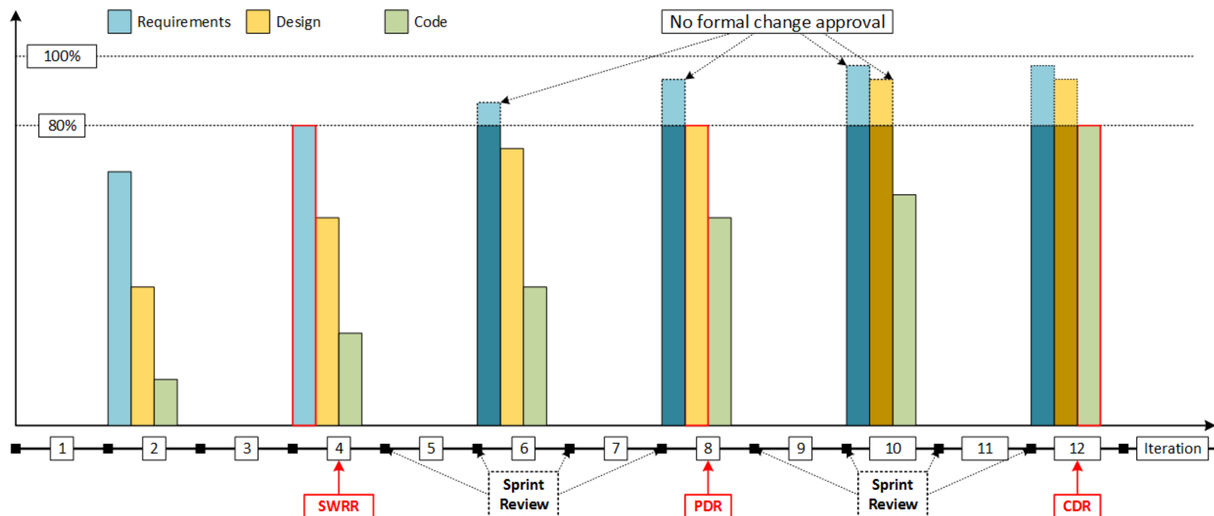

Figure 6-4 Model 2: More flexible review driven lifecycle

6.6.2.4 Model 3: Review driven lifecycle with full flexibility

Tag line: Formally approved changes are allowed anytime

The schedule for the review is not necessarily imposed from outside. The mechanism of delta or versioned reviews is used to organise the agile process (e.g. to align with product releases). The difference to model 2 is that the change process is formalised through explicit delta version reviews.

Objective

Ensure that changes to requirements and architecture are formally approved by all the relevant stakeholders. Changes are allowed at any time, but have a relatively large impact in terms of resources

Advantages:

— Full flexibility

— Software is fully endorsed by all stakeholders as all changes are subject to review

Neutral

— Fully working software produced at each sprint. This reduces the likelihood of not meeting the requirements. On the other hand, every change is subject to approval and this could end up in work being redone.

— Reviews are difficult to coordinate with stakeholders (due to a potential large amount of them). Coordination at system level is therefore doable but complex

Disadvantages

— Potentially large number of reviews

Description

The reviews can be fixed in advance at regular intervals or called on demand. Reviews are repeated in the form of "delta reviews". They contain exactly the same level of formalism as the original ECSS-E-ST-40 reviews. Work during the sprints follows the usual agile approach, but the outcome is subject to approval during the formal or delta reviews. Additional requirements and changes can be added to the system at any time. The level of formalisation is high while retaining full flexibility. The main disadvantage of the approach is that it is demanding in terms of resources.

Difference to previous model

As in models 1 and 2, this model produces highly formalised outputs that can be agreed by all parties. These outputs, however, can be changed at any time and are subject to formal approval. This model represents a trade-off between a flexible approach and the need for formalisation at the expense of an increase in effort.

Suitability

Recommended when the requirements can be changed during the development if required but need to be formally agreed. A good example is a scenario where the requirements or the architecture need to be prototyped in order to ensure that they can be implemented.

Variations

It is possible to have pre-reviews instead of delta reviews

It is also possible to combine delta reviews together (e.g. delta PDR + delta SWRR)

This model allows the adoption of the concept of the pre-reviews: depending on the project needs, the sprints leading to a joint-review can be anticipations of that review, in form of pre-reviews. This allows achieving a desired level of maturity before conducting a join-review (for example, 90% of the architecture is fixed before PDR).

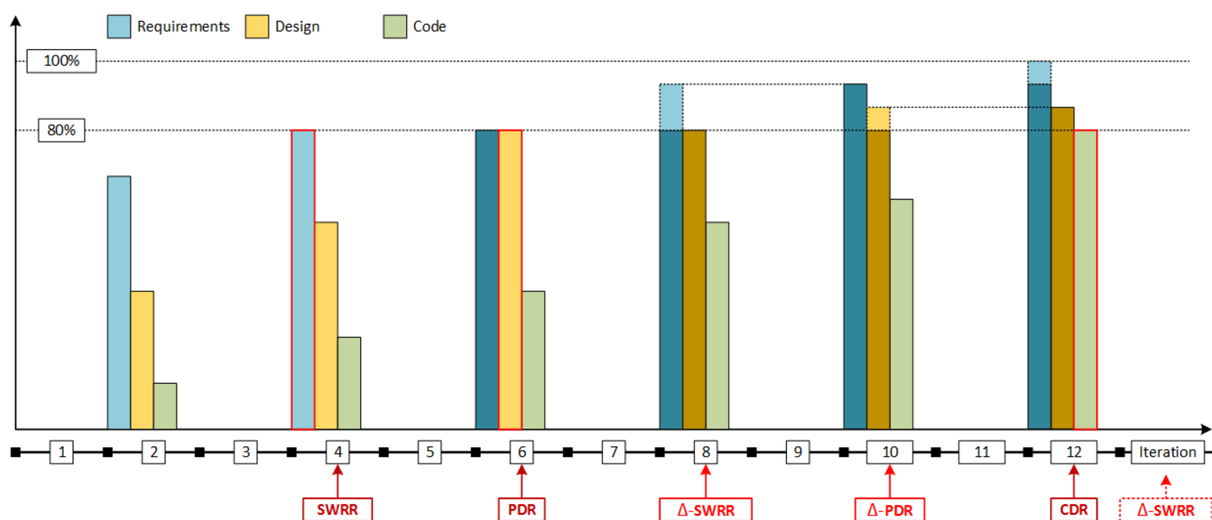

Figure 6-5: Review driven lifecycle with full flexibility

6.6.2.5 Model 4: Sprint driven lifecycle with formalisation

Tag line: Informally approved changes are allowed

Objective

Ensure that changes to requirements, architecture and code are (formally) approved by a reduced number of stakeholders. Changes are allowed after each review. The impact is somehow limited, but the level of formalisation is reduced.

Advantages:

- Full flexibility
- Full working software produced at each sprint. Since no approval is required for the changes, the development is unlikely to incur in wasted efforts.
- Formal reviews are limited in number.

Neutral

- Coordination with system development is limited to formal reviews and does not take place for the whole development
- The formal endorsement by stakeholders is only partial

Description

The reviews take place at regular intervals. Reviews are repeated in the form of "one single delta review". The level of formalism is reduced in the sense that stakeholders can delegate their attendance, some practices can be waived and some documents may be removed or partially produced. Work during the sprints follows the usual agile approach, although the outcome is subject to approval during the reviews, it is unlikely to be rejected. Additional requirements and changes can be added to the system at any time. The level of formalisation is reduced but flexibility is high.

Difference to previous model

As in model 3, this model allows unlimited changes subject to approval at the review. The main difference between both models is that the reviews in model 4 are implicit (while in model 3 are explicit). This basically means that the reviews are considered to be embedded within the agile lifecycle regular activities. These implicit reviews will typically involve a limited number of stakeholder and therefore should be less demanding in terms of resources. Some of the typical review requirements and practices are expected to be relaxed or downsized.

Another important characteristic of this model (shared also by model 3) relates to the split of effort among the different processes. In both models 3 and 4 the three primary processes are run from the very beginning almost in parallel, while in models 1 and 2 there is some weighted distribution of effort to reflect the sequential order (i.e. more work in initial sprints on requirements and architecture and moving towards implementation in later sprints)

Suitability

Recommended when the requirements can be changed during the development and require a certain level of discussion but no formal agreement

Variations

Variations of this model can in extreme cases (e.g. for pure prototyping activities and some of R&D activities) combine some or all of the reviews and/or hold all of the reviews as implicit reviews covered through sprint reviews.

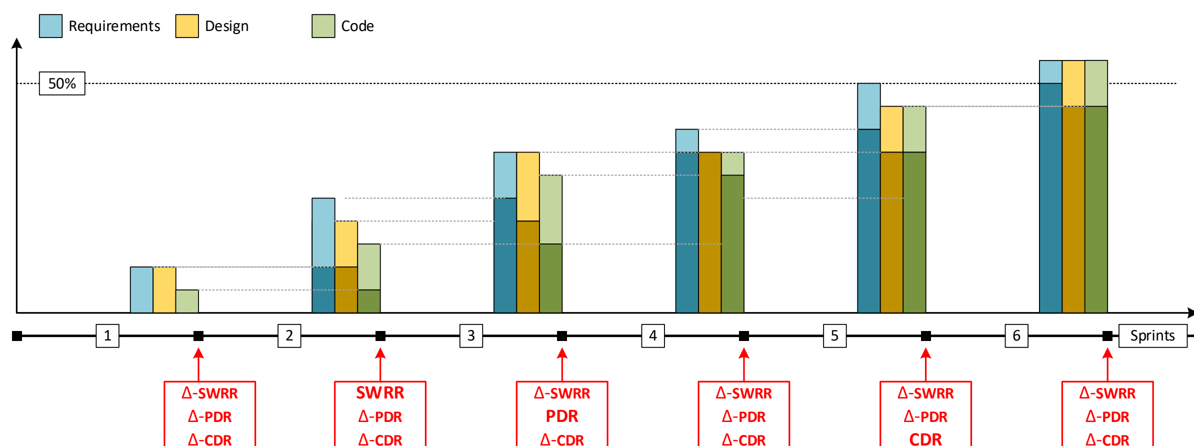

Figure 6-6: Sprint driven lifecycle with formalisation

6.6.3 Selecting the right model

Some of the factors discussed in clause 5.2 can be used as starting point to select one of the models described. More specifically, the customer context and the project context are the most relevant ones since they determine the required degree of formality that needs to be demonstrated regarding compliance with ECSS-E-ST-40.

The model descriptions provide the main charecteristics for each model and give hints towards situations where a particular model may be more or less suitable for a project.

Table 6-2 provides some examples herefor. When strong formalised coordination with higher level work is required, model 1 and model 2 work well, given that they offer a limited number of formals reviews. If such coordination requires a lesser degree of formality (e.g. via the product owner or scrum team), models 3 and 4 may be a better fit. Formal endorsement of requirements is best achieved using models 1 and 3, as any changes will be agreed in formal reviews. Flexibility needs, are well addressed by models 3 and 4. Model 4, in particular, is the most flexible option at the expense fo less formalisation.

Table 6-2 – Examples of selection of models based on project characteristics

Project characteristics	model 1	model 2	model 3	model 4
Strong coordination with higher level team via reviews				
Strong coordination with higher level team via agile				
Formal endorsement required				
Requirements need to be flexible				

Some orientative examples for each model are given below

- Model 1: Development based on an existing reference architecture and well known stable requirements and technology. This limits the risk posed by changes.
- Model 2: Development based on an existing reference architecture and known requirements but without detailed knowledge of potential technologies to be used. In such case, some room for changes is needed
- Model 3: Development involving external parties, where standardised interfaces are critical and changes may be required but need to be formally agreed.
- Model 4: Development based on new or unknown technology with target requirements, where it is possible trade off performance by cost (e.g. relax requirements to achieve cheaper solutions).

7 Guidelines for software project management

7.1 Introduction

Previous clause 6 introduced a reference model for Scrum-like software developments and has shown several options for implementing an ECSS compatible lifecycle. This clause aims at providing guidelines for software project management for Agile projects. The Software Development Plan (SDP) is the central document in the ECSS-E-ST-40 standard that requests a description of project management activities. Consequently, this clause 7 is organised in accordance to the document requirements definition of the SDP (DRD Annex O of ECSS-E-ST-40).

7.2 Software Project Management approach

7.2.1 Overview

Clause 4 has introduced the key Agile principles: the Agile Manifesto and the philosophy of lean management. These are the drivers for managing Agile software projects. We now detail these by providing guidelines for Agile software project management.

7.2.2 Management objectives and priorities

7.2.2.1 General

Classical V-model or waterfall-based processes, as presented in ECSS-M-ST-10 clause 4.4, are based on the assumption that in projects the costs, the scope, and the schedule of the project can be well defined. The practice shows however that for many software development projects these assumptions are often not met. The motivation behind following the Agile methodology often is that one of these aspects is not always clear. Figure 7-1 illustrates this in the project management triangle. It describes that flexibility and agility can be achieved by making one or more of the dimensions of scope, schedule, and cost flexible and keeping the others static. This flexibility is needed to be able to react to potential changes.

7.2.2.2 Cost Schedule Scope triangle

Project situations with both fixed cost and fixed scope should be avoided, as this minimizes the possibilities for being agile. The simple reason is that if both the cost and the scope are fixed, the only variable things are quality and schedule. The alternative is to have fixed-cost, variable-scope projects.

Here, cost, quality and schedule can be fixed, and the variable dimension is the scope. The scope can then be detailed with the customer. The rest of clause 7.2 describes how to cope with static and dynamic aspects of the cost-scope-schedule triangle.

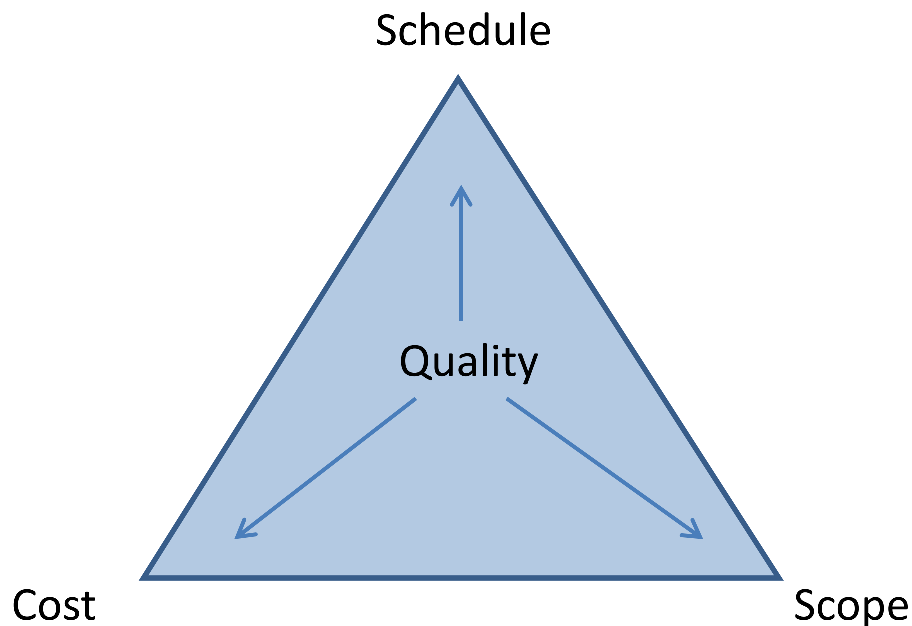

Figure 7-1: Project Management Triangle

The scope of an Agile project can be managed by using the means that are described in clause 6, like prioritizing the product backlog or applying the Product Owner role. This effectively manages changes with respect to the scope of the project. The scope of an Agile project can be described using epics, user stories, requirements, or other issue types for describing user needs.

The schedule of an Agile project can be managed by following the Agile methodology using the reference model that was introduced in clause 6. Defining the roadmap of a project using sprints and having commitments to these allows for a proper schedule management and ensures the flexibility to react to changes and to reschedule activities.

Cost management techniques for Agile projects help to introduce flexibility for the cost related aspects and to cope with agility. The most popular techniques are "Money for nothing" and "Change for free". The latter is described in the following clause 7.2.2.3 as an example.

7.2.2.3 Cost Management: Change for Free

Given that the customer is constantly participating to the Scrum team during the entire project, the concept of change for free enables them to make changes to the project scope. The customer needs to participate by being available for the discussion of user needs (for example requirements or user stories) and the prioritization of the backlog. These customer changes are accepted without requesting additional costs. Figure 7-2 illustrates this concept. Features (or requirements or user stories) are prioritized and ordered by business value and implemented in sprints, as usual in Agile projects. The customer requests a new feature X to be implemented; this feature has a higher priority and business value than another feature Y that is implemented later. The new feature X is accepted for free by the project team at sprint boundaries. Consequently, the other feature Y is removed from the backlog. This concept is valid if and only if both features X and Y have the same complexity and effort. Otherwise

maybe more features need to be de-scoped. Effort estimation guidelines are described in the following clause 7.2.2.4.

Both customer and the supplier should agree on how to deal with changes to the backlog (scale of complexity, relative assessment...). Change should always be considered in a positive manner and not as a drawback.

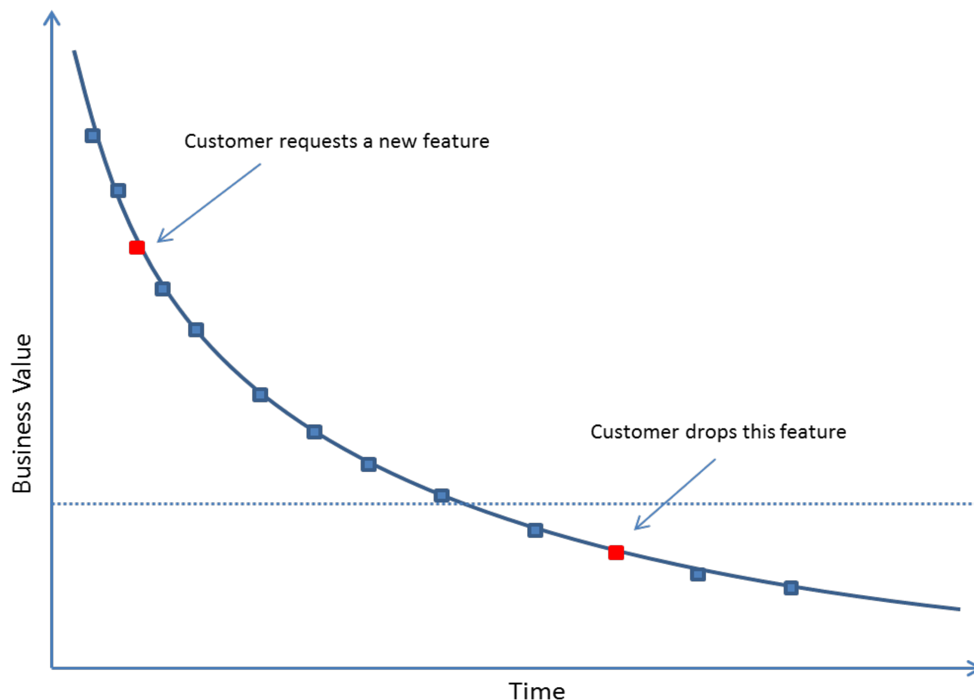

Figure 7-2: Cost Management: change for free

7.2.2.4 Cost Management: Money for nothing

The customer can stop the contract at the end of any sprint, subject to a penalty which amounts to a percentage of the remaining budget (e.g. 20%). The customer usually takes this decision when the cost of continuing the project is higher than the value that will be obtained.

The company doing the work is committed to execute a percentage of the work (e.g. 80%) to the quality agreed (definition of done) and on the agreed delivery date.

A requirement of the money for nothing approach is that the customer must participate in the team regular meetings.

7.2.2.5 Effort estimation

7.2.2.5.1 General

Proper effort estimation is of major importance for every development activity. Agile techniques use specific means to estimate the efforts. This is due to the fact that several new concepts can be incorporated in the estimation. Compared with the reference model that was introduced in clause 6, this includes the use of epics and user stories, the prioritisation in the backlog, roles like the Scrum Master or the Product Owner, or the Agile meetings.

Agile development projects are often built using the concept of integrated teams. This implies that classical estimation techniques are of limited use. Other estimation means are necessary to properly

allocate resources to the different activities. The Agile community knows a couple of effort estimation techniques that are summarized in the following clauses 7.2.2.5.2 and 7.2.2.5.5.

7.2.2.5.2 Story points

Agile project teams often use relative measurement units instead of absolute ones like man-days, man-hours, or ideal developer days. Instead, the unit that is used often is story points. Story points compare relatively user stories, epics, or requirements (see clause 8.3.1), for instance by assigning elements of the Fibonacci-series to them. This allows for their relative comparison and is used to measure the velocity of a sprint. Compare clauses 7.2.3.2 and 7.2.8.2 for details on measurement and control details.

7.2.2.5.3 Planning poker

Planning poker is a consensus-based gamified technique towards estimating efforts like story points for user stories in Agile development. The development teams use playing cards with Fibonacci-numbers (0, 1, 2, 3, 5, 8, 13, 21, ...) to estimate each user story. In estimation meetings, they play these cards face-down to the table, instead of speaking them out loudly. Cards are then showed by the team at the same time (to avoid biasing from peers and peer pressure), most extreme estimates are further discussed to allow for further clarifications, testing estimation assumptions and generating increased knowledge of the team over the subject. These estimates are then discussed and a consensus is agreed for each story.

7.2.2.5.4 Magic estimation

Similar to planning poker, magic estimation uses planning cards to estimate user stories. Here, the playing cards are put on a table, sorted by the effort they represent. Then each team member puts printed-out requirements (or user stories) on the table and places them below the playing card that represents the estimated effort for this requirement. Then the entire team walks around the table and compares the efforts and discusses changes and differences.

7.2.2.5.5 Ideal developer days

The overall absolute effort of a project can be measured by summarizing story points and multiplying them with an absolute factor, that can be based on man-days or ideal developer days (IDD). Experience has shown that there is a significant gap between a man-day and the work that can be done on a day. So an ideal developer day can be compared with a man-day by using a certain factor like 0.x. It has then to be adjusted that a story point can be treated as one or two IDDs, which then are two or three man days.

However, since the relation between velocity measurement and sprint effort (in man days) is not linear (due to various factors like team cohesion, team mastery, organizational entropy, etc.), an absolute factor based on an ideal developer day cannot be accurate. The overall absolute effort of an agile project can also be measured by adding-up backlog story points divided by the current velocity measurement and multiplying them with a constant sprint effort (in man days).

7.2.2.6 Make relevant decisions

From classical project management it is well known that decisions need to be made in every project. Formal decision processes like Kepner-Tregoe or decision trees exist to support making decisions. Agile projects often follow the Lean methodology as introduced in clause 4.1.3; here the focus is on lean values when making decisions:

- The customer value is the most important decision criteria, putting the customer demands in the focus of the process.

- Most decisions are made as late as possible. Some decisions need to be made early, other decisions do not need to be made, and most decisions are made as late as possible to ensure that most of the aspects of the decision are known.
- Remove Waste: a formal decision process often is not needed. It is needed in cases where catastrophic risks have to be avoided, significant schedule delays can occur, or significant cost increases need to be prevented. In most other cases, decisions can be made on a daily basis without following a formal process using the concepts that are introduced in clause 4, like using the planning meetings as a forum or by executing the Product Owner role. Here, the Product Owner decides on a couple of things, e.g. in terms of functionality management and prioritization.

7.2.2.7 Functionality prioritization

There are several methodologies to allocate priorities to the different functionalities of the system under development. The MoSCoW-method is a prioritization technique that divides functionality (referred to as "requirements" in the method) into Must, Should, Could, and Won't "requirements" as follows:

- **Must**: "Must requirements" are implemented in any case to guarantee the success of the project. If at least one "Must-requirement" is not implemented, the software delivered by the project is considered as a failure. MoSCoW is often used to select which requirements (epics) must be delivered in a minimum viable product (or MVP). All those epics marked as "Must" are implemented in order to deliver the MVP.
- **Should**: The "Should-requirements" are important but not necessary for the delivery. They can get more importance later or can be discarded.
- **Could**: The "Could-requirements" are desirable but not necessary. They can improve the user experience or customer satisfaction and are typically taken into account when time and resources allow.
- **Won't**: The "Won't-requirements" are typically agreed with the customer as to represent the least-critical functionality with the lowest business value. They are often not planned into the schedule.

7.2.3 Schedule management

7.2.3.1 Schedule definition

Agile schedules are often defined based on features and product releases, not against tasks that will be executed. The schedule focuses on delivering capabilities and features towards the customer, being it end users or the system level. An Agile schedule then is made up of a roadmap that shows the features that are delivered and implemented in sprints.

Schedules often change over time to align with evolving user needs. The Agile methodology provides the mechanism to incorporate the change in the development lifecycle.

The focus on sprints and features makes it necessary to have all skills in the project. Compared to classical projects, there are fewer disruptions in terms of people assignments, due to the fact that team members have to commit themselves to sprints and their goals. On the other hand, it is possible to assign people to other projects after the completion of a sprint. On the other hand, it is possible to assign people to other projects after the completion of a sprint, but there is a risk to have a major impact of the collective knowledge and skill of the team, which at the end will result on performance degradations.

Major milestones are included in the schedule, like the delivery of major versions after n sprints or the link to the formal ECSS reviews (see clause 6.6). The sprint goals can be aligned for these milestones, for example the preparation of the review. An alternative is to keep these two levels of sprints and ECSS milestones separated, as described in clause 6.6.20.

A common practice to prepare for the software delivery for a major release is to apply a 3-2-1 sprint sequence: in the case where a usual sprint lasts three weeks, two sprints that last two weeks and one week can be introduced to mature the release. The shorter sprints then have the goal of identifying bugs. No new functionality is introduced in these sprints; only in-depth-testing and bug-fixing are allowed.

7.2.3.2 Schedule control and performance evaluation

The evaluation of the performance of the schedule is a key activity in Agile projects and usually done by the team itself. The major outputs to measure are the 'delivered business value's in terms of working software. Teams do this with the use of Agile performance metrics. Most popular metrics are burndown charts, an example is shown in Figure 7-3. The green line shows the ideal burndown of all the efforts in a sprint. These efforts can be made up of user stories, tasks, features, etc. The blue lines show the remaining efforts and tasks. In faster times, the lines are below the green ideal line, in slower times the lines are above. The bottom line shows the completed tasks, per day. Combined with the concept of story points (not shown in Figure 7-3), Agile teams can count the number of story points they "earn" per sprint. This is an influencing factor for the schedule, because this creates an understanding of how many work, being it hours, story points, business value, lines of code, or other metrics, can be "burned down" in a sprint. This gives a good prediction means for the future performance of the next sprints and is often referred to as the velocity of an Agile project. This velocity is often given by the number of points per sprint. This is important information for the team since it gives them a good understanding of their own performance and is often used in planning meetings for the commitment to work that can be achieved by the team in the sprint.

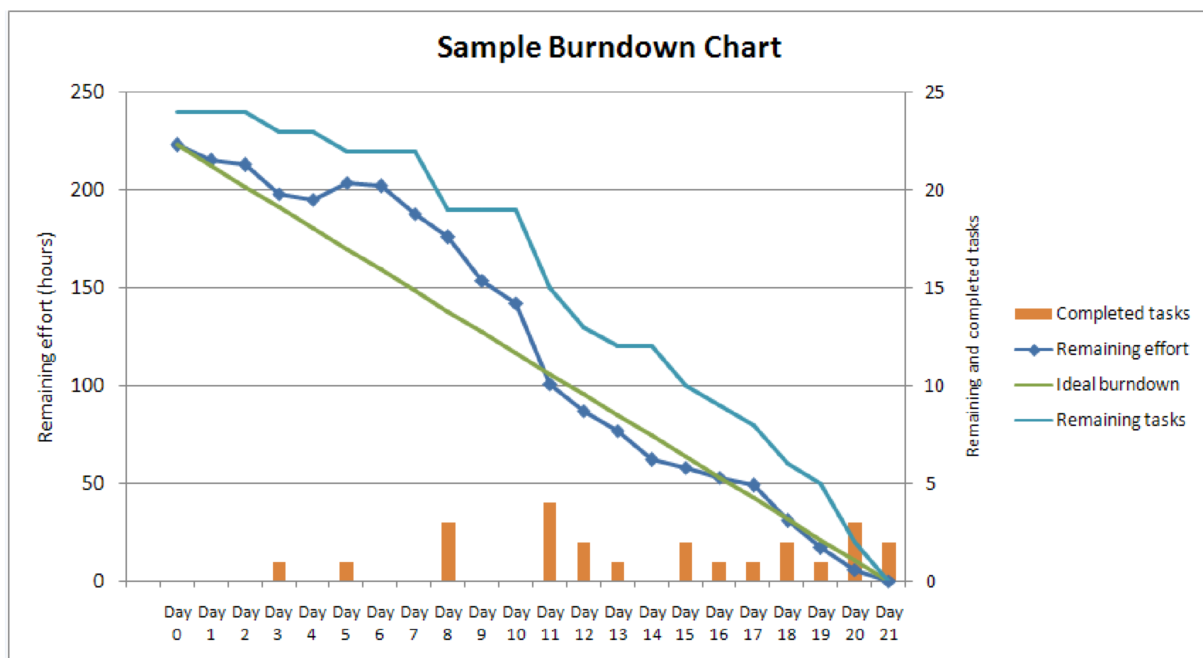

Figure 7-3: Sample Burndown Chart for a Sprint

7.2.3.3 Schedule reporting

Clause 7.2.2.4 has highlighted some of the Agile metrics like story points or business value. These metrics are customer-oriented metrics in the way that they take the customer needs into account and make them most important for the project. The reporting of the schedule performance takes these metrics as a basis. Modern Agile tools support these metrics and the creation of burndown charts and can be used to provide an insight into the schedule performance and can be presented towards management and the customer. This reduces the effort of collecting data and creating reports, as in classical projects.

7.2.4 Assumptions, dependencies and constraints

The software development plan defines the approach towards managing a project. For Agile projects, this includes defining how to concretely execute the Agile methodologies. This handbook defines reference models for Scrum-like development; the SDP defines the concrete implementation and highlights the major assumptions, dependencies, and constraints that need to be detailed for Agile projects. These are as follows:

- Selection of one of the models that were described in clause 6.6.2 on organizing the ECSS reviews in an Agile project.
- Description of the Agile SW development approach and lifecycle: several concrete implementations of Agile developments exist, so needs to be detailed whether a more full Scrum method is followed, whether a Scrum of Scrums approach is executed for larger projects, or whether other concrete methods, like Kanban, are followed.
- Scheduling: the schedule is described in the plan, in order to be Agile and to keep flexibility for changes, it needs to be decided whether a concrete sprint plan needs to be defined or not and how sprints are aligned to major milestones.
- Tool support: Agile development principles like continuous build, test, and delivery heavily rely on proper tool support and automation. This is explained, especially for SW Test, SW V&V, documentation, and reporting purposes.
- Agile requirements management: Agile software development introduces several concepts of handling requirements, like using Epics or User Stories. It also describes means to prioritize and process them. Managing user demands and inputs can be done in user expert groups as part of the product owner activities. This is described and linked to project management activities like customer interaction or milestone and review planning.
- Generation and automation of Documentation: Agile software projects aim at delivering SW early and often. This implies that for the necessary documentation an approach is needed how to iteratively document the project and the SW. This can be based on using generated documentation, lean documentation templates, and iterative documentation approaches where only the implemented parts of a SW are described in a design document. The chosen strategy is always agreed with the customer.

7.2.5 Work breakdown structure

Agile projects break down and structure their work according to delivering the final product. Clause 7.2.2.4 has already introduced that the classical organisation of a WBS is of limited use, since teams are often organized as collocated multi-skills teams that focus on a given technical perimeter, instead of being decomposed into tasks.

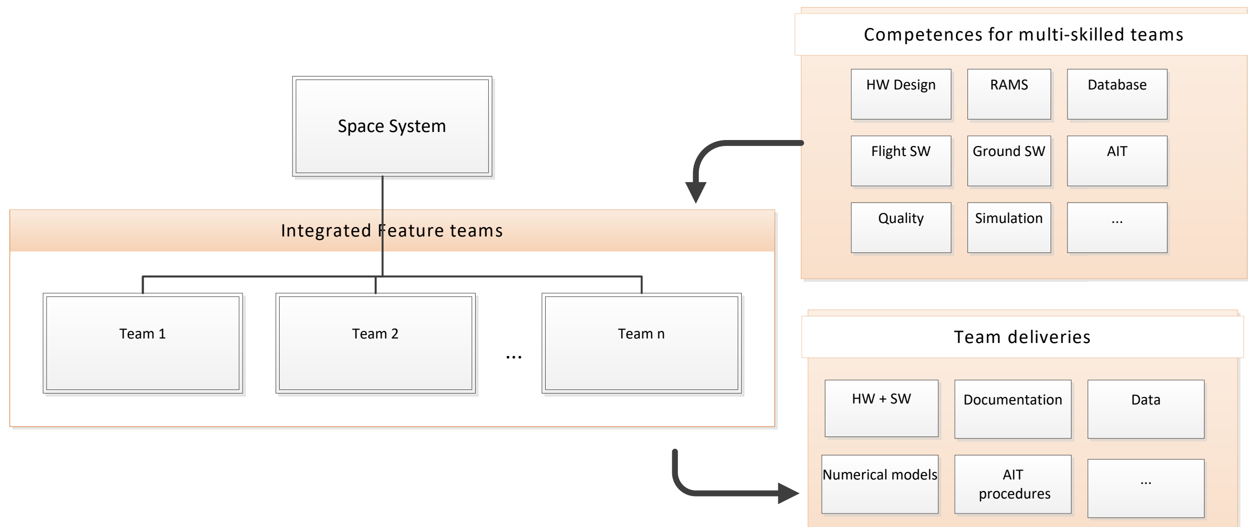

Figure 7-4: Example for an Agile work breakdown

Figure 7-4 shows an example for an Agile work breakdown structure, where a product "Space System" is decomposed into "n" integrated feature teams. These teams need competences and deliver products. Regarding competences it is ensured that all necessary skills for the teams are available. Each team can focus on a given technical domain like data handling, avionics, or diverse subsystems.

From an industrial point of view, supplier (or subcontractors) can be included in this approach by making them an integrated part of the teams, or by providing separate teams. This supplier handling is described in greater detail in clause 7.2.11.

7.2.6 Roles

The Agile method introduces several new roles, like the Scrum Master or Product Owner. It is useful to define own work packages for them to have the budget allocation and task definitions clear. It is possible that other roles like the classical project leader or team leader are not necessary, so it is important to make the concrete setup explicit in the work breakdown. It is important to understand that these roles do not introduce "more" people; instead these roles can be mapped to existing people like the project leader. The organisational breakdown often includes having functional teams, cross-functional teams, or transversal teams as introduced. The selected organisational structure needs to be described in the SDP.

7.2.7 Risk management

7.2.7.1 Overview

Risk management (described in ECSS-M-ST-80) is about identifying, collecting, and mitigating project risks and is a well-known technique in classical project management. The Agile reference model that was introduced in clause 6 supports risk management with various means that are shown in Figure 7-5 and are described in this clause.

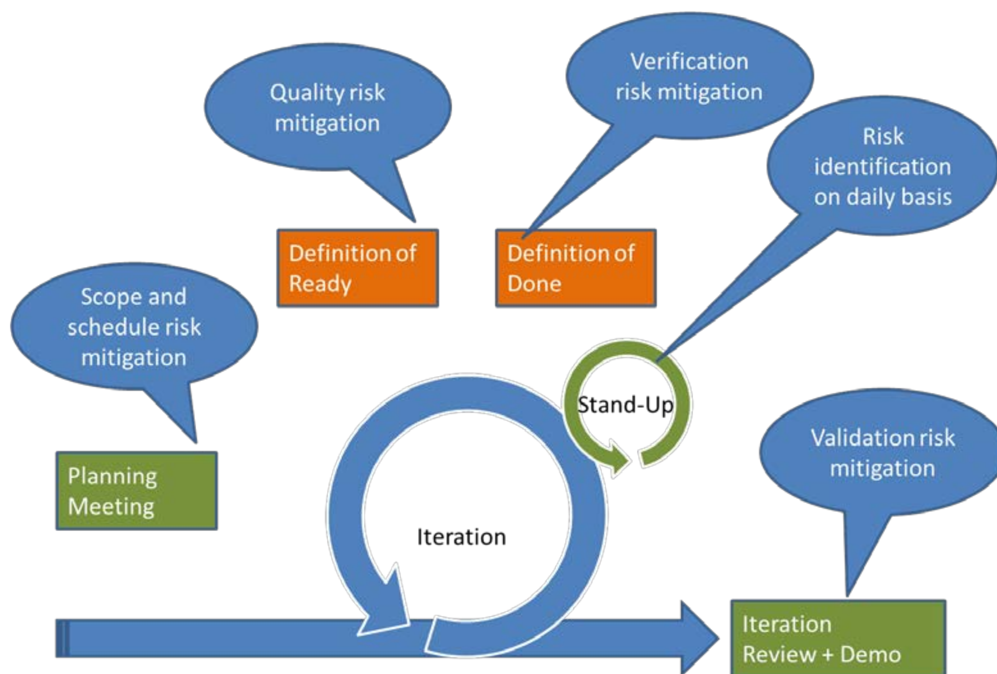

Figure 7-5: Agile model supports risk management

The identification of a new risk is the first step in risk management. The Agile methodology supports this with the Agile meetings: the daily stand-up meeting is the first source of emerging risks that are identified by the development team. Here, the Scrum Master is responsible for the collection and further communication of the raised risks. The sprint review meeting is the source of new risks in terms of technical risks like not meeting the Definition of Done and not satisfying functional or non-functional requirements. The sprint demo meeting is about mitigating the risk that the wrong product is built. The customer and the project team ensure that the implemented functionality satisfies the customer needs. This is a verification activity of managing that the right thing is built.

Verification risks are mitigated by the Agile method: the Definition of Done ensures that the product is built correctly. Consequently, quality risks are mitigated in the way that in every sprint each work product fulfils the agreed Definition of Done. This allows for a very early detection of verification risks.

Technical risks focus on violating non-functional requirements like not meeting performance or memory requirements. Two Agile means mitigate this risk. The first Agile technique focuses on giving non-functional requirements the highest priority in the product backlog to ensure that these are implemented first. This implements a fail-fast approach and clearly highlights the risk of not meeting these requirements. It is noted that anyhow, non-functional requirements are persistent qualities and constraints that, unlike functional requirements, are typically revisited as part of the Definition of Done (DoD) for each Iteration or release. The second technique is that Agile projects often start with a

prototyping phase. This is a good method to identify and mitigate technical risks by analysing technical issues.

Schedule risks can be identified early and mitigated properly in Agile development projects. The execution of sprints makes the team velocity transparent and the future performance can be well predicted. Potential schedule delays can come from various sources like technical debt, unavailable resources, or untrained staff. The Agile method makes this explicit and the risk can be identified at an early point in time.

7.2.7.2 Risk sources

Typical risk sources include the following risks: on supplier side, the Agile methodology can introduce the risk of doing more than it was estimated, so this results in a cost issue, also having an impact on the schedule. On customer side, the Agile process can have an impact on the schedule, delivering the product too late, for example because of the implementation of more features than expected. This can also have a cost impact by running out of money before having all must-have functionality implemented. Another risk source on customer side is quality; here it is ensured that the software is working properly after final acceptance. Documentation contains another risk: it needs to be ensured that it is not postponed for future work. Fluctuation in the team can also be a severe risk; the Scrum methods recommend to avoid this.

The described risk sources are serious risks, but a proper application of the Agile method can help to mitigate these. Concretely, the Definition of Done, the role of the Scrum master and the product owner can contribute to risk mitigation.

7.2.7.3 Risk process

Due to iterative nature of the agile process and the constant planning and review activities, certain risk related to schedule, quality, cost and scope are more likely to be detected earlier through the development lifecycle. While this handbook does not deal with basic risk handling procedures, it is clear that the agile approach offers the opportunity to raise the risks and react (if necessary handling them following the formal risk procedure) as soon as they are discovered.

7.2.8 Monitoring and controlling mechanisms

7.2.8.1 Communication and reporting

Communication is a key asset in every project to ensure proper knowledge and information distribution and management. The nature of Agile projects eases the communication by introducing the stand-up meetings. These meetings help the development team to exchange knowledge between them. Larger projects need to do these stand-up meetings on more levels, for example by having stand-ups on system level, on subsystem level, or by each function or discipline. Concrete guidelines cannot be given due to the diversity of projects, but a project has to ensure proper communication.

The communication with the customer is done on a regular basis by having him involved in the Agile sprints. Here the customer participates in the planning of the sprint and the reviews, as described in clause 6. More formal communication are aligned to the milestone definitions and documentation requirements from the ECSS-E-ST-40 and ECSS-Q-ST-80. Here it is important to align the milestones and documentation needs to the nature of the Agile project. Often, a full Software Design Document cannot be provided to a certain milestone review due to the fact that the software is implemented iteratively and that the design is only available for the already implemented software.

Monthly meetings on project management level can also support the Agile project, as in classical projects to report on the status and progress to the major stakeholders, being the customer, the program management, the line management, and the development team. These meetings are critical to monitor the expenses and resource consumption.

The lean management philosophy explains that the creation of reports is a kind of waste and is avoided. Instead, reports can be automated and extracted from the project specific tools like issue tracking systems and quality tools. The recommendation here is to use the same reporting formats and intervals for all stakeholders. Good experiences have been made with Wiki based reporting and the reporting on technical progress, the risks and their status, major decisions, the economic and the schedule status. With respect to the schedule control as described in clause 7.2.3.2, it is recommended that the reporting is based on the technical performance, instead of classical milestone performance tracking.

7.2.8.2 Measurement and analysis

7.2.8.2.1 Overview

Several Agile metrics exist that can be used to support measurement and analysis for Agile projects. These metrics can be divided into metrics for the team, the project management, and the customer. The most popular metrics are described in the following clauses.

7.2.8.2.2 Team metrics

Especially the team internal measurement and analysis relies on displaying and analysing metrics in real time for immediate feedback, like the success of the continuous integration test, the comment coverage, unit test coverage, or the violation of coding rules. Several tools exist that support the measurement of these metrics, their visualisation, and the immediate analysis by the teams. Figure 7-6 shows an example of the success of continuous integration tests, including the history of failures and successes.

Retrospectives are important meetings that provide both qualitative and/or quantitative feedback. Most Agile teams are able to capture and provide concrete performance metrics over the sprint based on modern tools.

The collection and analysis of team metrics can be supported by dashboards and systems that collect metrics. Figure 7-7 shows an example of code metrics including the evolution of lines of code, comments, test success, and code quality metrics.

All | Burrow | Dashboard | Gate | Glance | Keystone | Milestone-proposed | Nova | OpenStack-CI | Openstack-manuals | Overview | Quantum | Swift | Websites

S	W	Name ↓	Last Success	Last Failure	Last Duration
		glance	15 hr (#89283)	5 days 15 hr (#89277)	2 min 3 sec
		glance-merge	15 hr (#116)	1 mo 2 days (#78)	3 sec
		glance-pep8	15 hr (#289)	16 days (#269)	6.5 sec
		keystone	12 hr (#400)	13 hr (#399)	1 min 45 sec
		keystone-merge	12 hr (#292)	6 days 15 hr (#267)	4.6 sec
		keystone-pep8	12 hr (#335)	6 days 15 hr (#310)	9 sec
		keystone-pylint	12 hr (#408)	6 days 15 hr (#383)	22 sec
		nova	3 hr 44 min (#121729)	19 hr (#121717)	8 min 19 sec
		nova-merge	3 hr 44 min (#215)	21 hr (#194)	1 min 0 sec
		nova-pep8	3 hr 44 min (#1588)	21 hr (#1567)	1 min 9 sec

Figure 7-6: Success of continuous integration tests

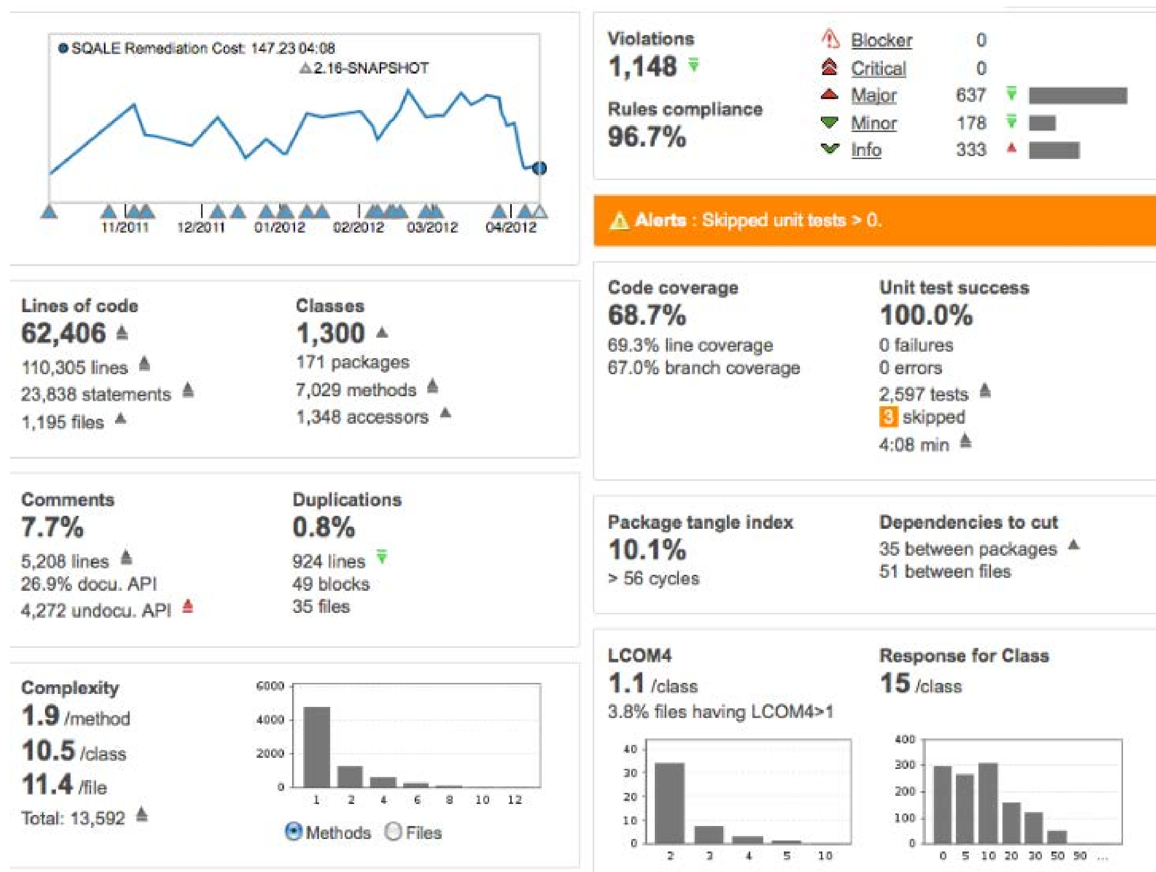

Figure 7-7: A team metric dashboard

7.2.8.2.3 Project management metrics

The project management usually is interested in the velocity of the Agile teams. Several metrics exist for supporting project management, like a distribution chart of bug fixes, implemented requirements, and delivered software. Figure 7-8 shows an example chart that compares the size of the backlog (red) compared to the work that has been performed (blue). It shows that the velocity of the team is almost stable and linear. The chart also shows that the backlog is actively managed, i.e. it is maintained and backlog items are continuously created. This is a good indicator that the task of requirements engineering is performed and that the team will not run out of work.

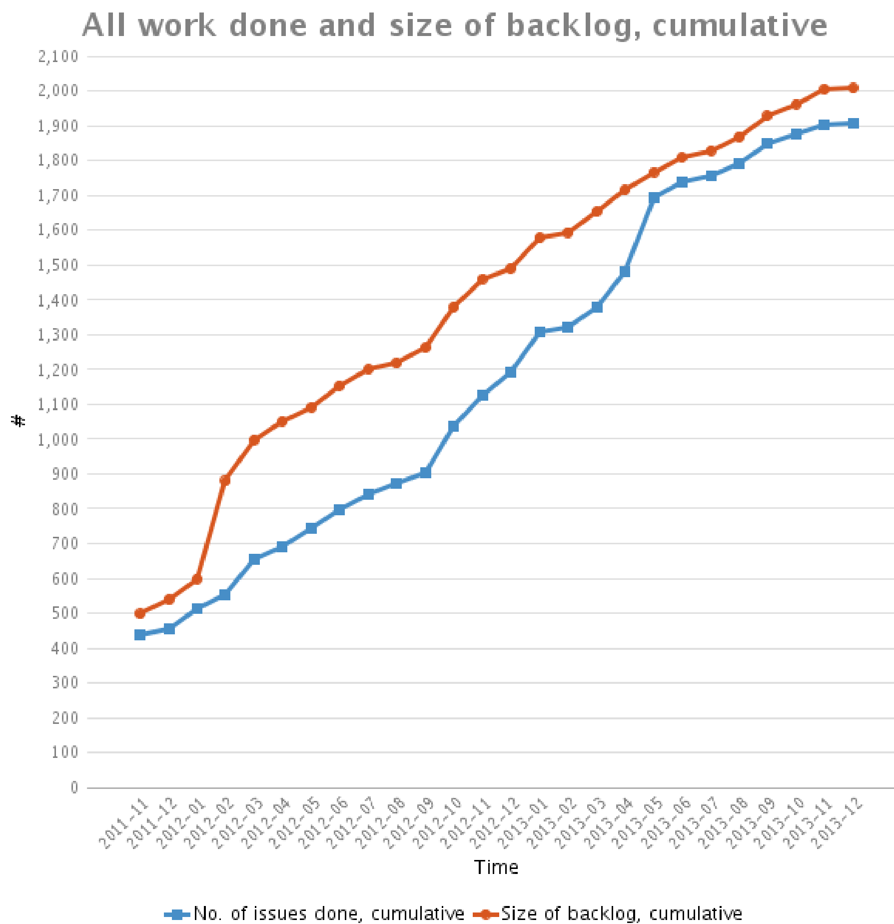

Figure 7-8: Summary of performed work

7.2.8.2.4 Customer metrics

The main focus of the customer is the delivered business value. This can be considered as the number of user stories that have been implemented, or as their amount of story points, for example. This can be done for projects that have a large system as a customer as well as for projects that are directly used, not embedded in a system context. Figure 7-9 shows an example chart that shows the delivered business value for a given project. The measure for the business value is defined first; it can be made up of story points, for instance.

Another kind of customer metrics comes from measuring the satisfaction of the customer. Customer satisfaction surveys can be performed on a regular basis, e.g. with a questionnaire that is answered by the customer. Another means for this is the net promoter score; it focuses on the question whether the customer is willing to promote the project team or supplier to colleagues or other projects.

Figure 7-9: Delivered business value in a project

The described metrics need to be defined and properly analysed. The analysis of metrics can be done using a separate process or in the Agile process. The retrospectives, the reviews and the daily stand-up meetings can be supported with the described meetings.

7.2.9 Staffing Plan

7.2.9.1 Overview

The well-defined organisation of a project team is a key requirement for the success of any team, being it Agile or not. The Agile specific organisational aspects are described in the following clauses, allowing to set up a team and to scale it regarding the project size. This includes setting up small projects as well as large projects, defining the necessary communication and reporting structures, and to set up the needed measurement and analysis structures.

Team building for Agile projects has to respect some Agile principles and is scaled for the size of the project. The Agile reference model that was introduced in clause 6 can be scaled to fit for smaller projects as well as for large projects.

7.2.9.2 Agile team building principles

Agile teams commit themselves during the sprint planning activities (see 6.3.2 and 6.3.3) to a certain amount of work that they can achieve. The concrete work assignment can be done by assigning work to each individual team member or by empowering the team to let themselves decide on how to implement what by whom.

Cross functional teams are often built in Agile projects. This means that the entire Agile team have all necessary skills that are needed to build the sub-products, product or system. Therefore an Agile team

can be made up of generalists as well as specialists. It is important that no single resource for a certain skill exists, but that skills are shared within the team. This has an influence on the teams' commitment to the sprint if a certain skill is unavailable, e.g. due to vacation or illness.

Common or collective code ownership describes the concept that the entire source code of a project is owned by the entire development team in terms of responsibility. This allows any team member to make changes to any part of the source code. This ensures that all team members have a good understanding of the source code and that every piece of code can be optimized and refactored by anyone.

Agile training is necessary for the successful installation of an Agile team. It is important that the Agile mindset is well known by the development team and that it is accepted and followed. External coaching can be applied for unexperienced Agile teams to ensure the proper application of the method.

Clause 6.2 has already described the major Agile team roles like the development team, the Scrum Master and the Product Owner. For the successful team building it is important that the roles have been understood and properly assigned to individuals. Experience has shown that it is important that the development team is protected against disruptions, so that they can fully commit themselves to the project. Experience has also shown that often some roles are merged, e.g. a Scrum Master can be a developer. In general, however, it is not advised to have the same person playing the role of product owner and scrum master as their objectives are somehow not compatible. These combinations have advantages as well as disadvantages. It is important that these constraints are understood before the team is set up.

7.2.9.3 Agile for large projects

The Agile methodology that was introduced in clause 6 can be applied for small projects that consist only of one Scrum team of 5-9 team members. It can also be applied for large projects, where the project is made up of a couple of teams. This approach is often described by the Scrum of Scrums method or variants. Existing solutions are made of creating several kinds of teams, like having feature teams, having dedicated test teams or having tester in every team, having requirements teams, or having other team setups.

7.2.9.4 Pair programming

Agile software development knows about the concept of pair programming, which is about having a pair of two developers working together on one workstation. One of them writes the code, the other one reviews it on-the-fly. Usually they share and swap the roles frequently. This method is more costly on the first view, but studies and industrial experience show that the created source code has much fewer defects. Other advantages are that pair programming supports team-building and communication and is a good supporter of knowledge management by distributing knowledge within the team.

7.2.9.5 Peer reviews

Another technique for knowledge distribution and management in Agile teams and for identifying defects are peer reviews, where team members review the code their peers have written. This needs to be managed and can be directly mapped to the product backlog items (PBI). In line with the concept of the Definition of Done (see clause 6.3.11), each PBI can be peer reviewed before it is finally changed to "Done". This concept is not limited to source code; it can also be applied to tests, design, user stories, and other work items.

7.2.9.6 Skill management

Development teams are often requested to be cross-functional. These teams are multi-skilled in the way that they have all necessary skills that are needed to develop a specific piece of software. For example, it is possible that an avionics development team needs to have HW design, RAMS, database, ground SW, and simulation skills to perform their work. Here it is important to distinct between the Scrum team and the development team. The latter implements the software and needs to have the skills; a skill matrix is strongly recommended and beneficial to ensure that all skills are covered. The Scrum team includes the product owner and the Scrum master.

Experience has shown that Agile teams not only need technical skills. Soft skills also become more and more important. Abilities of discussing design alternatives, moderating creative thinking sessions, performing root cause analyses, and facilitating and participating in stand ups and retrospectives are necessary skills for Agile teams and their members. It is possible that it is necessary to spend a certain effort to ensure that all development team members are trained w.r.t. soft skills for Agile development.

7.2.10 Software procurement process

No specific activities are performed for software procurement for Agile development. However, specific activities can exist in order to have a collaborative approach to agile contracts (for example, pre-commitment phase, early exit, change for free).

7.2.11 Supplier management

The management of suppliers can be divided in two views, focusing on the industrial view with subcontractors and the customer view, like agencies.

Agile supplier management from an industrial point of view can be divided in two popular scenarios, being integrated teams and non-integrated teams.

Typically, Agile projects create integrated teams with their suppliers. Here the suppliers provide work force, tools, hardware, and knowledge and integrate in the teams on customer side. This approach has a couple of benefits: the risks are actively shared with the suppliers by having them committed to sprint goals; the communication between customer and supplier is optimized by having the supplier fully integrated in the Agile process; the monitoring and control of the supplier can be done on a low level, linking success to the delivery of business values and establishing trust. Knowledge can be shared on a daily basis within the development team by performing daily work, pair programming, or peer reviews.

Non-Integrated teams do not integrate their suppliers in the development team. This calls for more project management efforts regarding the synchronisation of the supplier and the customer. User stories are flown down for implementation to the supplier. Milestones are defined as integration points where implemented software can be integrated in the overall software system. The overall strategy is defined regarding full or partial integration, or regarding participation in the continuous build and test strategy of the project.

Agile supplier management from an agency point of view with agencies as a customer mainly focuses on the management of the supplier in terms of managing the project. Here, it is important that the Agile approach is supported by the customer, i.e. by participating in the sprints and their reviews, and also by assigning a product owner on customer side to the project.

7.3 Software development approach

7.3.1 Strategy to the software development

The overall strategy to the Agile software development is described in the SDP. Clause 6 describes a reference model for Scrum-like Agile developments. Its concrete application in a real project is defined in the SDP.

7.3.2 Software project development lifecycle

The software development plan describes the software project management cycle, according to Annex O of ECSS-E-ST-40. Clause 8 describes guidelines for Agile software engineering processes. The SDP details these processes for a given project.

7.3.3 Relationship with the system development lifecycle

Clause 6.6 describes several models for the integration of Agile software developments in system contexts. The concrete implementation of the approach for a given project is described in the SDP. This description needs to take into account the implications on communication, reporting, synchronisation of changes, as well as the logic of scheduled deliveries, their acceptance reviews, and their verification and validation.

7.3.4 Reviews and milestones identification and associated documentation

Clause 6.6.2 introduces several models for the alignment of the ECSS reviews and the described Scrum-like Agile software development lifecycle. Clause 6.6.3 gives guidelines for selecting the proper model for a given project.. All these information are taken into account when defining the review and documentation plan for a project. For the selected model is described how a given project concretely follows this model, how documentation is created, and how reviews are organized. This is agreed between the customer and the supplier and its description is part of the Software Development Plan.

7.4 Software engineering standards and techniques

The Agile and Lean approaches introduce additional engineering techniques. Clause 4 provides an overview of the state of the art. The SDP details how these are applied and how they are in line with ECSS-E-ST-40 and ECSS-Q-ST-80 requirements.

7.5 Software development and software testing environment

The Agile approach calls for a software development and test environment that supports continuous change, integration, and test. Clause 9.1.8 describes some of the basic features for such environments. The SDP further details the concrete setup within a real project.

7.6 Software documentation plan

In general an evolving documentation approach is a key feature of Agile projects. This implies that documents may not be in the final version at a certain milestone and are living documents and are continuously updated at each sprint (see models in clause 6 for details). One possible approach is to limit the content of the living documents to the functionality that has been already implemented. This has some implications on the documentation planning. The important issue is that the documentation approach is agreed between supplier and customer.

The possibility of frequent deliveries has an implication on the creation of documentation. The clause 4.1.3 introduced with the Lean philosophy that the effort for document creation is minimized. As a consequence, most documents can be created automatically, or be simply represented by live and updated snapshots of "the electronic format of a tool database, to be agreed with the Customer" (as mentioned in ECSS-E-ST-40 Annex R).

Formats like wiki-pages, extracts and reports from issue-tracking systems, and data in general can be treated as documents. A requirements baseline for instance can be made up by using the product backlog.

The creation of documentation can be a key component of the Definition of Done (see clause 6.3.11). This means that the agreed documentation is created continuously and that an implemented functionality is set to "Done" only if the necessary documentation is available, being it documented in a wiki, as data, as source code comment, or in other ways.

An alternative approach for the creation of documents is to have dedicated tasks for it. These tasks are then separate entries in the backlogs.

The review of documentation can be discussed on two levels: it can be peer reviewed during daily work as part of the regular reviews of the Definition of Done. The other level is a formal review with the customer; here the expectations on Agile documentation are clearly defined.

For formal reviews, clause 6.6.2 has introduced different models on how to align the ECSS reviews with sprints. The creation of documentation takes into account the selected model. For instance, for model 1, where ECSS reviews are not part of the sprint lifecycle, the two ways of creating documentation (continuously as part of the Definition of Done of a user story or separately in dedicated tasks) are balanced. For the dedicated document creation alternative it is decided how this effort is managed and scheduled, i.e. with a separate sprint that focuses on ECSS review preparation and document creation.

The same is true for the other models, it is decided whether documentation is created as part of the Definition of Done or whether this is a dedicated task.

8
Guidelines for software engineering processes

8.1 Overview

Clause 8 discusses independently the different Software Engineering processes involved in an Agile development. Taking into account a Scrum-like lifecycle as defined in clause 6, it elaborates on the aspects of adopting this lifecycle on each engineering process in detail in the following clauses.

8.2 Software related system requirements process

The main objective of the software related system requirements engineering process (ECSS-E-ST-40 clause 4.2.2) is to produce functional, performance and interface requirements.

For best practices we distinguish between two cases:

1. Either the RB is produced following the process defined in ECSS-E-ST-40 clause 4.2.2. without any specific aspect of Agile lifecycle taken into account. In this case the RB is an input to the agile software development lifecycle (e.g. typical for on-board software developments)
2. Or there is no sharp distinction between system, user and software requirements, hence RB and TS can be merged and the agile process starts from populating the Backlog with epics, user stories and tasks (which are described in the next clause).

8.3 requirements and architectural engineering

8.3.1 Software requirements analysis

The Software Requirements Analysis is one of the two activities of the software requirements and architecture engineering process. The logic of this activity is based on the two main concepts of the Requirements Baseline, which is typically produced by the Customer and the Technical Specification, which is typically produced by the Supplier.

The RB is often composed of the system and user requirements. The TS is the response to those user requirements in form of detailed software requirements. The border line between the user and software requirements is not sharp and there is usually some level of overlap between the two. For ground software development, which is based on heavy reuse of infrastructure (reference architecture), it is not unusual that the customer defines directly the software requirements (SRS) as a starting point of the software development activity.

The main motivation behind the steps involved in the Requirements Analysis Phase is to discuss, agree, document and baseline the requirements before starting with the design and implementation of the software. The underlying concept of waterfall software development approaches is to "Avoid Change". In this mindset it is very important to capture the specification of the software in detailed, precise and complete requirements as early as possible at the start of the software development lifecycle. As a result a lot of effort is dedicated to capturing the complete set of requirements, reviewing these through a formal process and documenting the results.

Experience of many software development projects has shown however that specifying a complete set of detailed and quantified software requirements is often a very challenging when not impossible task. This is in particular true for performance and quality requirements. Especially when developing a new type of software application, the conceptual view of the software, as it is perceived by the users at the time of specification, can be very different from the actual resulting application.

The Agile software development methodologies have incorporated change as a natural part of the lifecycle rather than an "anomaly". In the reference Agile software development lifecycle, which is outlined in clause 6.5, the concept of user and software requirements are accordingly mapped to typical Scrum constructs such as Features, Epics, User Stories and Tasks. It is important to appreciate the different approaches that Agile software development methodologies take towards solving the challenge of specifying the desired software functionality before it is implemented and communicating the customer needs to the supplier. The difference is not in how the specification constructs are named or structured. It is much more on the fact that by incorporating the change into the nominal process (not as an "anomaly" but as the standard step in the process) the motivation and the focus of the software requirements analysis activity is completely shifted from trying to capture everything upfront, hence avoiding (or limiting) any possible future change. In Agile methods the motivation of the same activity has moved towards adapting and improving the requirements throughout the software development process. This way the results of implementation of a subset of requirements by the supplier are demonstrated to the customer and the feedback taken into account during the subsequent software analysis activity for the next phase. In other words, the purpose of demonstrating the intermediate results and feeding back the comments is to improve the requirements.

Despite the difference in the naming and format of the constructs, it is possible to provide a mapping between the concepts of Agile software development methods such as Features, Epics, User Stories and Tasks on one side and the user and software requirements artefacts of ECSS-E-ST-40 on the other side.

Epics and User stories best map to user requirements as they have a user oriented focus. The formulation of user stories in the format of "As a Role X I want Y in order to achieve Z" is a good indication for this mapping. When it comes to non-functional aspects of the software, it is recommended to amend the corresponding user stories with software quality attributes such as performance.

As a guideline for the granularity of user stories, a user story is implemented in one sprint (one development cycle). A group of related user stories which together aim at a higher-level objective/goal can form an epic, which maps to high level user requirements.

Specifying user stories in the format prescribed by the Scrum methodology (i.e. beginning by "as a Role I want to … in order to ") could not be practical in some cases. For instance, when the requirements come from the system or when reporting errors to be corrected as part of the sprint.

User Stories are then broken into tasks, which are annotated with effort allocations (estimates, updated later with actuals). Tasks are good candidates for mapping to software requirements. However, there is not always a one to one mapping, as a task can capture a requirement on a team member ("Install DB on Server xyz"). It is therefore suggested to split the project management tasks

and those related to functionality of the software under development. Only the latter are relevant to the ECSS-E-ST-40 Software Requirements Engineering process (clause 5.4.2). Figure 8-1 shows the example of a user story (left side) and its broken-down to five constituting tasks (right side).

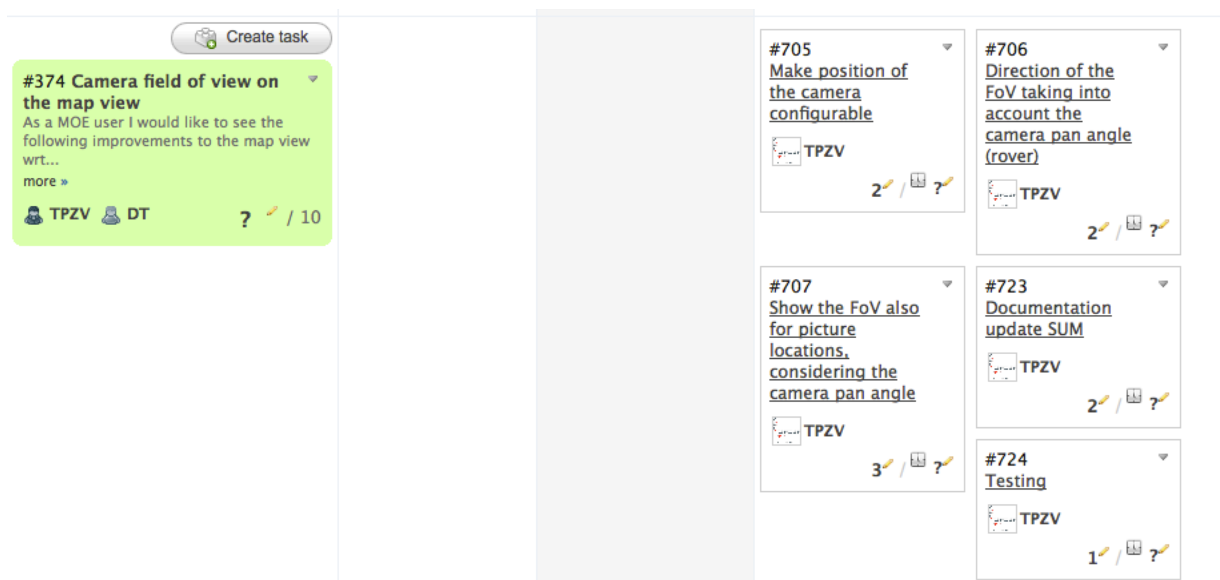

Figure 8-1: Example of User Story and Tasks

The first sub-activity in the software requirements analysis activity as specified in ECSS-E-ST-40 is Establishment and documentation of software requirements.

The outcome of the software requirements analysis activity is the Software Requirements Specification, SRS. Having a SRS is mandated by the ECSS-E-ST-40. The format and content of the SRS is however project specific. The level of documentation has always been a point for debate. As a good practice the user stories contain enough details, so that they can serve as a baseline for the demonstration and validation. One of the corner stones of the Agile Manifesto is “Working software over comprehensive documentation”. To what extent this principle is embraced by a project depends on the nature of the software under development, the culture of the organisational unit and the non-functional requirements of the system, e.g. long-term maintainability. As there is a direct interaction between the product owner, the customer representative and the developers at the sprint planning meeting, the interpretation of requirements and their details can be discussed directly. Hence there can be less need for verbose and very detailed explanation of the requirements when using the Agile SDLC (unless other project needs, e.g. reuse of requirements or long term maintainability exist). Also at the end of the sprint, when the feature is demonstrated it helps in further specifying the remaining requirements (user stories) or refinement of already implemented requirements for the next sprints. Now, to what extent this discussion is captured formally and documented, either by updating the wording of the User Stories and the associated tasks or in Technical Notes or in complementary MoM depends on the project needs.

The product backlog is the repository of the Features, Epics, User stories and tasks and can be best mapped to the RB and Software Requirements Specification. It is important to appreciate that the product backlog contains inherently the traceability of software requirements to user requirements (e.g. as shown in Figure 8-1). This traceability is given by breaking the user stories into tasks. A user story representing the user requirement and the tasks representing the software requirements, which are driven from that user requirement. The same way, the tractability of high level user requirements can be tracked by mapping them to epics and features and breaking them down to user stories.

For some user requirements and the related software requirements, such as the ICDs in response to IRD, the backlog can only contain the user requirements but the actual ICD is a separate document, which is generated as a result of a task of the associated user story or stories.

One of the common misconceptions about Agile methodologies is that Agile methods are suitable for projects, for which the requirements are unknown. For a successful Scrum-like project, it is essential to start with a solid product backlog, which captures the details of all user stories as specific as possible to the best knowledge of the users at the start of the project. In practice this means to spend as much as possible the effort to specify user stories at the very beginning of a project and create a baseline SRS. The flexibility of the Agile process can be used to improve the quality of the requirements and refine them gradually during the software development lifecycle.

The difference lies in the approach towards baselining the requirements. In the Scrum-like process, outlined in clause 6, the user stories for each sprint are reviewed and agreed by the team and the Product Owner at the beginning of each sprint during the sprint Planning (see clauses 6.3.2 and 6.3.3). It means the focus is put on agreeing on the subset of the requirements, which are relevant to implement particular feature(s) that is the target of that particular sprint. At the end of the sprint, the achieved results (the implemented feature) is demoed to stakeholders and their feedback is captured and propagated into the software development process, by updating some of the user stories in the product backlog, adding new user stories or removing existing user stories. The idea behind is to use as much as possible the feedback of the users, as the experience shows that the users often like to revise their initial requirements, once they see how some of them are implemented. As a result, the product backlog can be conceived as a living SRS, which is becoming more and more reviewed and agreed on as we run through the sprints. If the ECSS review Model 3 (introducing delta reviews for changed functionality, see clause 6.6.2.4) is adopted, the sprint planning can be considered as delta SRR/SWRR for each sprint. A Screen Shot of the sprint backlog (if a tool is used) can constitute the SRS portion for that delta review or an update of the Product backlog exported to a document can be generated, depending on the project needs.

In Model 1 (review driven lifecycle, see 6.6.2.2) of the reference software development lifecycle, at certain checkpoints during the project (e.g. after the 3rd sprint), larger portions of the product backlog can constitute a baseline SRS and be reviewed in a more formal SWRR and PDR joint-review. The planning of the SRR and SWRR can be either driven by programmatic schedule milestones or by technical considerations such as when all user stories related to the features of a sub-system are completed.

The Agile requirements engineering process heavily relies on "Deliver highest business value first". This means that these features with the highest value for the customer are implemented and delivered first. This allows the customers to already use the software and enables them to already fulfil their business needs. As a consequence, it is most important to have

a. a good understanding of the customer's needs, and

b. to know customer’s priorities.

Clause 7.2.2.7 briefly mentions the MoSCoW method for requirements prioritization and provides a definition of the terms. This method can easily be used to divide customer requirements into certain categories. These categories are made up of "Must", "Should", "Could", and "Won't" and indicate that requirements can be sorted into one of these categories. The idea behind is that all "Must" requirements have the highest business value and have to be implemented in any case to ensure the success of the project. In opposite, a "Could"-requirement is desirable, but not necessary. These requirements are an added value, for example for user experience or customer satisfaction, but are only taken into account for implementation if time and resources allow for them. Summarizing, the MoSCoW method is a good means for the prioritization of the user needs (requirements or user

stories) in the product backlog. It can provide a good input for the iterations to ensure that all requirements with the highest business value go into the first iterations, instead of burning resources for implementing "Could" or "Won't" requirements.

The Kano-model is another way of providing a view on the needs of the customer during the requirements engineering process. It goes back to the theory that a software system needs to provide all the "must-haves" or basic needs that are expected from the contracted system. If only these requirements are implemented, the system is accepted and ready for use. Additionally, the so called "delighters" are important as well to ensure the satisfaction of the users and customers and to finally guarantee the acceptance of the system. Figure 8-2 shows that "some" delighter features are sufficient to significantly boost the acceptance and satisfaction of the software. On the other hand, having only the "must-haves" implemented, does not dramatically increase the satisfaction. The backlog prioritization and maintenance process takes this concept in mind to ensure the final acceptance of the project or software system.

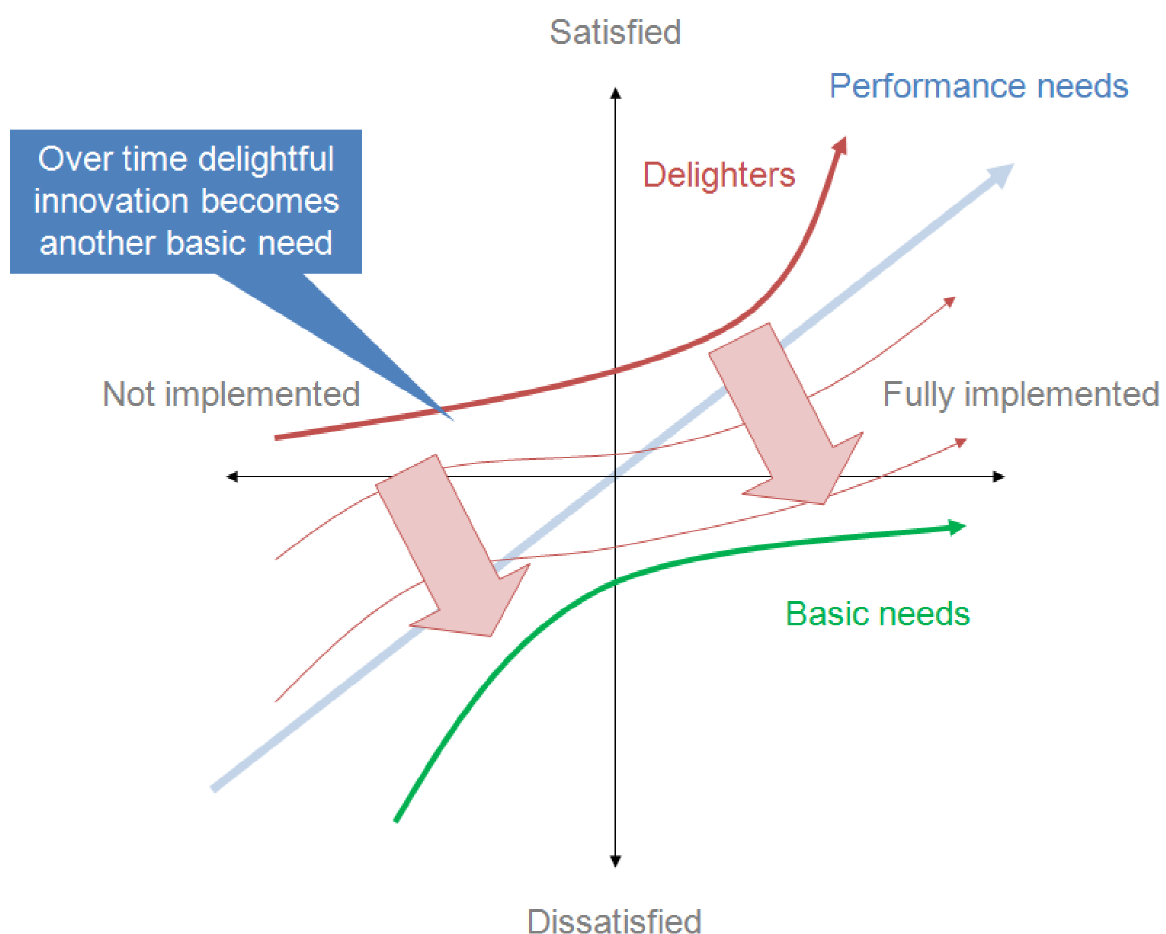

Figure 8-2: Kano model showing means to ensure customer satisfaction

ECSS-E-ST-40 requirements on Software Requirements Analysis can be traced to Agile activities as well as useful practices. Please refer to Table 8-1.

Table 8-1: Mapping ECSS-E-ST-40 to Agile activities. Software Requirements Analysis

ECSS-E-ST-40 clause	Agile activities	Recommendation
5.4.2 Software Requirement Analysis	Activity 1: Population of the initial backlog Activity 2: Review and refinement of the backlog	N/A
5.4.2.1 – Establishment and documentation of software requirements	Activity 3: Sprint planning	This is a repetitive activity in a Scrum-like lifecycle that is repeated at the beginning of each sprint. All requirements of the 5.4.2.1 are valid and are unchanged. The extent to which each of the expected outputs from a. to i. is specified, is independent from an Agile SDLC being adopted. This depends on the type of the application under development.
5.4.2.2 – Definition of functional and performance requirements for in flight modification	No specific activity. Part of sprint planning	N/A
5.4.2.3. Construction of a software logical model	Population of the initial backlog and sprint planning activity. No dedicated activity is needed when following a Scrum-like lifecycle.	All requirements of 5.4.2.3. remain valid. To what extent this is followed and explicit output is produced is independent from adopting an Agile approach, but much more dependent of the type of the software under development.
5.4.2.4 Conducting a software requirement review	Sprint planning activity Backlog refinement activity Sprint review (including demo)	Practically the software requirements review is broken down in repetitive activities of sprint planning and sprint review, repeated at the start and end of each sprint. Clause 6.6.2 provides five alternative models for handling of the reviews in the reference Scrum-like SDLC.

8.3.2 Software architectural design

The main objective of the Software Architectural Design process of ECSS-E-ST-40 is to transform the software requirements into an architecture by describing its main composing components, their internal and external interfaces, their dependencies and their run-time behaviour. The main products of this phase are the Software Design Document, SDD, and the Interface Control Document, ICD. Typically in Agile methodology the specification of the software requirements goes hand in hand with the architectural design.

The joint review, which marks the end of this process is the Preliminary Design Review, PDR.

In an Agile SDLC with evolving (not frozen) requirements, the architectural design (which is the transformation of those requirements into an architecture) inevitably evolves with the requirements.

This means in practice that the design of the software product is shaped and evolved gradually through the sprints.

It is however a good practice and a recommendation to devise a high level architecture of the envisaged software product as early as possible in the Scrum-like process, which is described in clause 6. This high level architectural design of the software decomposes the envisaged software product into its main components (like HMI, Database, Server logic, Client logic) and assigns the features of the Product Backlog to the main components.

It is also recommended to follow the best practice of specifying the external interfaces of the software product as well as the internal interfaces between its main composing components in form of ICDs as early as possible in the process.

It is not unusual in an Agile SDLC that during initial sprints, dedicated user stories and tasks are defined to perform trade-off analysis by implementing prototypes of the components to analyse the critical or unknown aspects of software. To give concrete examples, if the suitability (e.g. from performance or security perspective) of a certain technology or alternative design choices for implementing some elements of the software is not known, a prototype of that element can be implemented to assess it.

Where applicable, the real-time behaviour of the software is another aspect that needs to be assessed as early as possible in the SDLC. Hence it is recommended not to move these aspects and the related features to later sprints in the project.

The risk of not following the recommendations is that the features developed in early sprints may not meet the non-functional requirements of the overall software product and late architectural design changes can propagate back and require major re-design and re-implementation of already done user stories.

On the other hand, following the recommendations is likely to identify those requirements that are not needed or of lesser priority, once the impact on the architecture is explained early on to the customer.

The characteristic of the Agile methodology lies in the fact that it is understood and accepted by all stakeholders that the devised architectural design is not frozen and can evolve based on the feedback from the sprints, as user stories can be refined throughout the SDLC.

It is recommended to keep the SDD document as a living document.

With regard to the reviews, depending on the selected model described in clause 6.6.2, either each sprint meeting can be seen as a delta PDR, or a PDR can be conducted at a certain point in the SDLC (e.g. when 70% of the architectural design is well established and not likely to be changed significantly anymore). In this latter model the further changes to the Architectural design as a result of the feedback and discussions in the sprint meeting can be considered as delta PDRs in subsequent sprints.

In the third model of clause 6.6.2, a formal PDR can be hold as a joint review with other reviews at the end of the project.

With regard to software reuse, two aspects are taken into consideration. As the requirements can evolve, the choices for software reuse can evolve accordingly. Hence it is recommended to maintain the Software Reuse File as a living document. The second aspect is that due to reduced level of control over the reused part of the software, the scope of applying the Agile process on functionality provided by the reused software may be limited.

There is a close relation between adopting Agile methodology and continuous build and integration philosophy. This is mainly related to the motivation of Agile to provide working software throughout the SDLC as frequent sprint demos and product releases. Hence the choice of the tooling, procedures, schedule and the strategy that take into account this need and allow higher levels of automation.

ECSS-E-ST-40 requirements on Software Architectural Design can be traced to Agile activities as well as useful practices. Please refer to Table 8-2

Table 8-2: Mapping ECSS-E-ST-40 to Agile activities. Software architectural design

ECSS-E-ST-40 requirement	Agile activities	Recommendation
5.4.3.1 Transformation of software requirements into a software architecture	Adopting Agile has no impact.	Do it as early as possible. By sprint 3 a high level architecture is in place.
5.4.3.2 Software design method	Adopting Agile has no impact.	There is no preferred method. However the selected method facilitates frequent updates.
5.4.3.3 Selection of a computational model for real–time software	Adopting Agile has no impact.	There is no preferred method. However the selected method facilitates frequent updates.
5.4.3.4 Description of software behaviour	Adopting Agile has no impact.	There is no preferred method. However the selected method facilitates frequent updates.
5.4.3.5 Development and documentation of the software interfaces	Adopting Agile has no impact.	There is no preferred method. However the selected method facilitates frequent updates.
5.4.3.6 Definition of methods and tools for software intended for reuse	Adopting Agile has no impact.	There is no preferred method. However the selected method facilitates frequent updates.
5.4.3.7 Reuse of existing software	Adopting Agile has no impact.	N/A
5.4.3.8 Definition and documentation of the software integration requirements and plan	Sprint #0	N/A
5.4.4 Conducting a preliminary design review	See clause 6.6.2 – Agile lifecycle models	N/A

8.4 Software design and implementation engineering

Design practices in Agile indicate a so called "emergent" design, which infers that the software design is not specified fully up-front, but it evolves throughout the SDLC.

On an architectural level, in an Agile project, light-weight modelling is prepared at the beginning, which offers an overall vision of the architecture and its critical components. On sprint level, teams can perform more modelling as part of the planning activity in order to prepare for the upcoming iteration. If a TDD approach is preferred, then the software design is heavily characterised by the design of tests. Refactoring also plays a big role in Agile design and implementation, as it is a common practice for teams to make changes in the code without changing the semantics, in order to improve quality or prepare for the development of new components. Another fundamental building block of Agile software development is the continuous integration, which assures that new implementations or modifications do not break already existing functionality.

In the context of ECSS-E-ST-40, except for the production of the code, unit and integration testing of the software product, the Software Design and Implementation Engineering process outputs the detailed design of the software items, alongside justifications for design choices that meet the requirements. The resulting Design Justification File is taken as an input to a project's CDR.

As one of the principles of Agile is to cherish a "working product" over "detailed documentation" the level of documentation of the DJF depends very much on the project needs.

ECSS-E-ST-40 requirements on software design and implementation can be traced to Agile activities as well as useful practices. Please refer to Table 8-3.

Table 8-3: Mapping ECSS-E-ST-40 to Agile activities. Software Detailed Design, Coding and testing, and Integration

ECSS-E-ST-40 requirement	Agile activities	Recommendation
Detailed design of software items		
5.5.2.1 Detailed design of each software component	Planning I, II Product backlog refinement Coding, testing and documenting	Backlog refinement has an impact on the design, hence it is considered. Design changes and choices can be documented as part of the sprint.
5.5.2.2 Development and documentation of the software interfaces detailed design	Coding, testing and documenting	Interfaces are discussed and designed in parallel to the software components, during the same activities.
5.5.2.3 Production of the detailed design model	Coding, testing and documenting	The detailed design model can be created as part of the work of the sprint.
5.5.2.4 Software detail design method	Adopting Agile has no impact	The detailed design method is chosen by the team.
5.5.2.5 Detailed design of real-time software	Planning I, II Product backlog refinement Coding, testing and documenting	The design is created just as in the case of non-real time software.

ECSS-E-ST-40 requirement	Agile activities	Recommendation
5.5.2.6 Utilisation of description techniques for the software behaviour	Adopting Agile has no impact.	There is no preferred method. However the selected method facilitates frequent updates.
5.5.2.7 Determination of design method consistency for real-time software	Adopting Agile has no impact.	There is no preferred method. However the selected method facilitates frequent updates.
5.5.2.8 Development and documentation of the software user manual	Coding, testing and documenting	See clause 7.6
5.5.2.9 Definition and documentation of the software unit test requirements and plan	Sprint #0 Coding, testing and documenting	See clause 7.6
5.5.2.10 Conducting a detailed design review	See clause 6	N/A
Coding and testing		
5.5.3.1 Development and documentation of the software units	Coding, testing and documenting	This is the core part of a sprint's work.
5.5.3.2 Software unit testing	Coding, testing and documenting	Unit testing can be included as acceptance criteria and is therefore normally done before or in parallel with the development of software units. Practices: TDD, continuous integration, test automation
Integration		
5.5.4.1 Software integration test plan development	No dedicated Agile activity. Adopting Agile has no impact. Product backlog refinement	Test specifications can be added to user stories and tasks as part of product backlog refinement, if this is required by the project.
5.5.4.2 Software units and software component integration and testing	Coding, testing and documenting	High level of automation supports the frequent execution of this activity at each sprint.

8.5 Software validation

In ECSS-E-ST-40 the Software validation process is composed of:

- validation process implementation,
- validation activities with respect to the technical specification, and
- validation activities with respect to the requirements baseline.

The validation activity demonstrates compliance with requirements. Clause 8.2 explains the relation between the user and software requirements, respectively RB and TS, to the features, epics, user stories and tasks artefacts, which are typically used in Scrum-like lifecycle processes. When the project has explicit requirements, the user stories are normally mapped to them. The mapping can take place any time during the project activities or even before the start.

Model 1 (review driven lifecycle): testing is done outside the sprint lifecycle. For this approach the validation (specification of the test, performance of the test and documentation of the test) is completely transparent as it can be even performed by a separate team (that is not the Agile team).

Model 2 (review driven lifecycle with some flexibility): testing is done as part of the sprint lifecycle. This can be optimised by mapping some of the concepts related to demo, acceptance criteria for user stories and Definition of Done.

Where applicable, additional sprint tasks are defined to perform the following activities: map user stories to requirements, define tests, perform tests, document results.

At the sprint review the user story acceptance criteria are checked during the demo. This maps to validation with respect to RB in the ECSS-E-ST-40 (clause 5.6.4).

If required by the project, it is possible to validate at the level of tasks (or confirm that such validation has taken place during the sprint). This maps to validation against TS in the ECSS-E-ST-40 (clause 5.6.3).

The formal review and baseline of the test scenarios is performed according to the model selected by the project to perform the ECSS-E-ST-40 project reviews (see clause 5.5.2): the validation of user stories are systematically demonstrated and verified during sprint meetings in Model 3 (review driven lifecycle with full flexibility) and Model 4 (sprint driven lifecycle with formalisation), or during dedicated formal review meetings in Model 1 (review driven lifecycle) and Model 2 (review driven lifecycle with some flexibility).

In the Table 8-4 ECSS-E-ST-40 Software Validation requirements are presented and mapped to common Agile activities.

Table 8-4: Mapping ECSS-E-ST-40 to Agile activities. Software validation

ECSS-E-ST-40 requirement	Agile activities	Recommendations
5.6.2.1 Establishment of a software validation process	Sprint #0	N/A
5.6.2.2 Selection of an ISVV organization	Adopting Agile has no impact.	See clause 8.10
5.6.3.1 Development and documentation of a software validation specification with respect to the technical specification	Coding, testing and documenting activity Planning I and Planning II	See two models specified in this clause. This applies only to Model 2.

ECSS-E-ST-40 requirement	Agile activities	Recommendations
5.6.3.2 Conducting the validation with respect to the technical specification	Coding, testing and documenting activity	Only in case of Model 2 (including option b) Otherwise outside the Agile process.
5.6.3.3 Updating the software user manual	Coding, testing and documenting activity	Depending on the needs of project, the updates of the SUM can be produced at each sprint, each nth sprint or only for the "Product Release"
5.6.3.4 Conducting a critical design review	See clause 6.6	
5.6.4.1 Development and documentation of a software validation specification with respect to the requirements baseline	Planning I Backlog refinement	Only in Model 2 (review driven lifecycle with some flexibility)
5.6.4.2 Conducting the validation with respect to the requirements baseline	Sprint demo/review	Only in Model 2 (review driven lifecycle with some flexibility)
5.6.4.3 Updating the software user manual	Coding, testing and documenting activity	Depending on the needs of project, the updates of the SUM can be produced at each sprint, each nth sprint or only for the "Product Release"
5.6.4.4 Conducting a qualification review	See clause 6.6	

8.6 Software delivery and acceptance

As explained in clause 6.5, Agile sprints do not necessarily conclude with a working "software product". This means that a product release lifecycle is run in parallel to the Agile lifecycle. Each product release is typically associated to:

- A set of features
- A set of documentation
- A set of verification and validation activities

These elements are spread among several sprints and therefore there is the possibility that they are not validated in conjunction during the sprint reviews. For this reason it is advisable to plan at least one delivery and acceptance phase (as described in ECSS-E-ST-40 clause 5.7) to perform an overall validation of the product release.

The delivery and acceptance phase are aligned with the Agile development of each product release and foresee some effort to:

a. provide training and support as requested in ECSS-E-ST-40 clause 5.7.2.2,

b. correct potential problems, and

c. implement necessary changes or adaptions.

Although the basic requirements related to the software delivery and acceptance do not differ much from what is described in the ECSS-E-ST-40 standard, some aspects are more detailed. In particular, the following points need to be taken into account:

a. The installation process is likely to be fully automated in-line with continuous integration best practices, but it is likely that the platform is rebuilt from specification to ensure conformance to the baseline. Furthermore, it is important that the documentation describing the build and installation process are thoroughly checked.

b. The acceptance documentation is likely to require consolidation and completion before submission for review, since every user story modifies only the relevant parts. Similarly, the frequent software deliveries have an impact on the user documentation. Details on documentation handling are provided in clause 7.6.

In Table 8-5 ECSS-E-ST-40 software Delivery and Acceptance requirements are presented and mapped to common Agile activities.

Table 8-5: Mapping ECSS-E-ST-40 to Agile activities. Software Delivery and Acceptance

ECSS-E-ST-40 requirement	Agile activities	Recommendation
5.7.2.1 Preparation for the software product	Adopting Agile does not have an impact	Concepts of continuous build and automated deployment are closely related to the Agile-based SDLC, in particular when frequent Product Releases are planned.
5.7.2.2 Supplier's provision of training and support	Adopting Agile does not have an impact	
5.7.2.3 Installation procedures	Adopting Agile does not have an impact	There can be the need for it to be done without resorting to already existing continuous integration systems in order to validate the approach
5.7.2.4 Installation activities reporting	Adopting Agile does not have an impact	Possibly to be done in a lightweight manner (e.g. as sprint input)
5.7.3.1 Acceptance test planning	Adopting Agile does not have an impact	Can be based on existing acceptance criteria
5.7.3.2 Acceptance test execution	Adopting Agile does not have an impact	N/A
5.7.3.3 Executable code generation and installation	Adopting Agile does not have an impact	There can be the need for it to be done without resorting to already existing continuous integration systems in order to validate the approach
5.7.3.4 Supplier's support to customer acceptance	Adopting Agile does not have an impact	N/A
5.7.3.5 Evaluation of acceptance testing	Adopting Agile does not have an impact	N/A
5.7.3.6 Conducting an acceptance review	Impact discussed in clause 6	N/A

8.7 Software verification

In an Agile context, verification of the software is an integral part of every iteration, or sprint. All three activities defining the construction of a product – designing, coding, testing – as well as the ones related to them are performed together in short, repetitive cycles, where verification activities are embedded. The Definition of Done (DoD) can be used for forcing verification on multiple levels: design (e.g. design aspects have been reviewed by other members of the team), verification of code (e.g. compliance towards a coding standard, coverage metrics have reached their targets), unit testing (e.g. check that they have been performed and are consistent with the design and requirements), integration testing, and validation and acceptance testing. The DoD is imposed as a prerequisite to the sprint review. Failure to meet its criteria implies that the work is not done, shall not be presented to the Product Owner at the Sprint review and will negatively affects the team's velocity. Concretely, there is no acceptance of implementation without proving that all required verification activities have been performed.

Traceability requires a special mention. There are different levels of traceability: from requirements to design components, to the code and unit testing, to validation tests, to acceptance testing. In an Agile development approach, the requirements are not stable, therefore frequent changes have an impact on how the requirements are traced to other artefacts (e.g. design, code, tests). Some recommendations in order to implement traceability with Agile:

- Consider using the product backlog as the reference for describing traceability from requirements level, and not introduce yet another document.
- Since traceability can be an arduous effort to keep the trace items up to date, attempt to automate as much as possible. The features provided by modern Development Toolsets can facilitate traceability and establish links between artefacts.
- Decide which elements will be traced and between which levels, for example: explicit requirements (system level, product level) versus stories (user stories, technical stories) – see clause 8.3.18.3.1. Consider also the relation with epics, themes, tasks, etc. and how there can be a certain relation between them.
- It is recommended to wait until the third or fourth iteration before the traceability activity is initiated since sprints and releases do not need the same formalization level. This is because the requirements could not be very stable early on, which could be very disruptive when trying to establish a stable trace path. When the Definition of Ready and the Definition of Done are well established for a sprint or a release, it is foreseen that the development team is able to manage the traceability in the defined perimeter limited to a sprint or a release, in particular the traceability between stories and tests.
- If test-driven development (TDD) is applied, then effectively there is a direct and guaranteed trace between code to tests, since the test cases get created prior to the code.
- Some Configuration Management version control tools can help with establishing and maintaining traceability between the code and the executables, assuming version control on both. If version control is performed on test cases as well, then the connection with code and test cases can be established more easily.

In the Table 8-6,ECSS-E-ST-40 software verification requirements are presented and mapped to common Agile activities.

Table 8-6 Mapping ECSS-E-ST-40 to Agile activities. Software Verification

ECSS-E-ST-40 requirement	Agile activities and meetings	Recommendation
5.8.2.1 Establishment of the software verification process	Sprint #0 Retrospective meeting	Defining how the software is verified is something that needs to be set before starting the actual development, but it can also be decided on as an improvement in a retrospective meeting. Be it pair programming, peer reviews, these practices need to be understood and applied by the team. A Verification Plan can also be written.
5.8.2.2 Selection of the organization responsible for conducting the verification	Adopting Agile has no impact.	The development team is a self-organizing, independent body, responsible for all parts of the product, including nominal verification. See clause 8.10 for Independent verification.
5.8.3.1 Verification of requirements baseline	Planning I + II Product backlog refinement	User stories are kept up to date and therefore they are revised continuously. Additionally, before being taken into a sprint, a user story complies with the Definition of Ready defined by the team; this includes, among other items, an established set of acceptance criteria.
5.8.3.2 Verification of the technical specification	Planning II	The technical specification can be mapped to the task splitting of user stories. Verification of tasks is not a formal activity, but rather a team effort to find solutions to requirements (user stories) with the given know-how. The definition of acceptance criteria for each user story provides some kind of verification to user stories.
5.8.3.3 Verification of the software architectural design	Product backlog refinement Review meeting Ad-hoc meetings	Critical elements can be discussed at any given point throughout the sprint. Verification of the architectural design can be either done formally during the review meeting, or informally during backlog refinement meetings. The availability of the product owner is also an advantage as critical architectural details can be discussed at any time (and not just at formal meetings).
5.8.3.4 Verification of the software detailed design	Adopting Agile has no impact.	See clause 8.4
5.8.3.5 Verification of code	Coding, testing and documenting	This is a common software practice, and there are several practices used in Agile

ECSS-E-ST-40 requirement	Agile activities and meetings	Recommendation
		that aim code verification, depending on the project requirements. Practices: pair programming, peer review, unit testing, automated testing, continuous integration, support of the SDE.
5.8.3.6 Verification of software unit testing (plan and results)	Review meeting, Coding, testing and documenting	Results of unit tests can be addressed at any time during the sprint, but especially during the review meeting, depending on the criticality. Unit testing execution and validation can also be addressed as part of concrete sprints by defining the relevant user stories.
5.8.3.7 Verification of software integration	Review meeting, Coding, testing and documenting	Automated tooling is often used in Agile projects to make sure that there is always one working build available. Practices: continuous integration
5.8.3.8 Verification of software validation with respect to the technical specifications and the requirements baseline	Adopting Agile has no impact.	Parallel to the Agile processes, "requirements traceability", where teams can monitor which requirements/ user stories are covered by which test cases, what the status of the test coverage is, etc. This can be part of an established continuous integration process. Verification activities can also be programmed as part of a sprint by defining relevant user stories.
5.8.3.9 Evaluation of validation: complementary system level validation	Coding, testing and documenting	As part of the sprints, the whole system can be validated according to a given test plan. This implies integration of the sprint outputs into the system and it is recommended to align it with product releases.
5.8.3.10 Verification of software documentation	Review meeting	It is a common practice to include a checkpoint in the Definition of Done that enforces updates and creation of the software documentation. As part of the sprint review, the DoD is checked against every user story and therefore implies verification of documentation. It is, however, recommended to plan a final check of the whole document at some stage. Practices: Checking Definition of Done
5.8.3.11 Schedulability analysis for real-time software	Coding, testing and documenting	This is done as part of the sprint. In this purpose, simulators could be used. For OBSW, for example, a real-time test bench

ECSS-E-ST-40 requirement	Agile activities and meetings	Recommendation
		could be useful for such analysis.
5.8.3.12 Technical budgets management	Review meeting Product backlog refinement	Technical budgets, such as CPU utilization or memory size, can be established as a consequence of sprint results or refinement of the backlog, based on the needs of the ongoing project.
5.8.3.13 Behaviour modelling verification	Coding, testing and documenting	This is done as part of the work in the sprints, typically using a prototype.

8.8 Software operations

No specific aspect need to be said with regard to adopting Agile process.

The rationale is that the software operations process starts after the acceptance, which starts itself after the completion of the Agile software development lifecycle of each product release.

8.9 Software maintenance

8.9.1 Overview

Software maintenance activities can take place in parallel with the software development, when increments have been delivered and are used by the customer, and after acceptance by the customer, when the complete software is delivered and is operational.

8.9.2 Agile maintenance challenges

In some cases, mainly when maintenance is performed after software acceptance, the size of the team can be reduced to the minimum, such that the Agile process becomes irrelevant (e.g. when the maintenance team is composed of one single person).

Some maintenance mechanisms and drivers such as Service Level Agreements (SLA) or operational needs are triggers that need well defined schedules of activities that are hardly compatible with Scrum-like processes as outlined in clause 6. In such cases, other techniques are worth to be considered.

When the software production plan foresees several product releases or when intermediate products (e.g. sprint deliveries) are made available to the users of the product, software defects are reported along with user feedback during the development lifecycle. Fixing software defects therefore competes for the resources available in the team for the development of new user stories and features. One way of dealing with this is to allocate a high-level continuous work item for bug fixing or for the implementation of very urgent features. This continuous work item is assigned an envelope of e.g. 20%-30% of the sprint effort. Another way is to process these bug fixes and urgent features in independent branches of the development, to be merged within the main stream, e.g. for software releases. The selection and prioritisation of software defects to be fixed at each sprint can be handled either through the sprint planning I (what will be delivered, see clause 6.3.2) or, in cases of need for a very urgent reaction, this can be handled during planning II (how will it be delivered, see clause 6.3.3).

8.9.3 Tailoring Agile to Maintenance

8.9.3.1 Overview

The following subclauses show how the ECSS-E-ST-40 clause 5.10, "software maintenance process "can be mapped onto Agile activities. The Table 8-7 ECSS-E-ST-40 software maintenance requirements are presented and mapped to common Agile activities.

8.9.3.2 Process Implementation

8.9.3.2.1 Introduction

The process implementation is about the definition of the software maintenance process in the Software Maintenance Plan. The definition of the maintenance plan cannot be mapped to a particular Agile activity. Nevertheless, any project implementing an Agile maintenance process describes this process in the Software Maintenance Plan. The following clauses 8.9.3.2.2 and 8.9.3.2.3 describe the concerns that are addressed.

8.9.3.2.2 Sprint duration

In maintenance phase, sprints are not necessarily directly following each other. The **sprint duration** can be negotiated with the product owner. It is still time boxed but can be made shorter than during the development phase.

8.9.3.2.3 Maintenance categories in sprint planning

Different categories of maintenance are described in ECSS-E-HB-40 "Software engineering handbook" clause 5.10. Some parts of the maintenance can easily be planned, such as Planned Operational Maintenance, Preventive Maintenance or Adaptive Maintenance. They represent the need for planned activities aimed at preserving the system health. Each of these needs finds its place in the product backlog as a new story. New sprints are planned to implement these activities. This maintenance work is highly predictable and is easy to organise.

The management of unforeseen software problems and quick answers to new issues is called Unplanned Corrective Maintenance and is described in clause 9.1.6.

Evolutionary Maintenance often leads to significant additional development and is not necessarily achievable on one short maintenance sprint. Significant change requests can define several new user stories and are not always achievable in one short sprint. On the other hand, time constraints are also lesser. For these reasons, it is important to consider software modifications as regular software development and therefore it is important to follow the conventional Agile development cycle described in the preceding clauses.

8.9.3.3 Problem and modification analysis

In an Agile maintenance process, it is particularly important to define the correct kind of maintenance of a particular issue. User feedback from preceding sprints allows identifying new user needs and retrieving information about bugs and issues. This helps to add new stories during the product backlog refinement activity and to select stories and tasks for the next sprint during the sprint planning activity.

Product backlog refinement activity assigns priorities on stories and enables the maintenance team to estimate a working time. This activity is quite important during the maintenance phase. On top of creating stories, any new backlog entry is assigned to a maintenance category. Whenever possible, urgent unforeseen corrective maintenance can be introduced in the current sprint. On another hand,

important changes required by evolutionary maintenance can be handled in a dedicated development cycle outside of the maintenance cycle, maybe even by another team.

8.9.3.4 Modification implementation

Any modification or correction is implemented in the same way as in development phase. Agile guidelines described in the preceding clauses remain fully applicable.

8.9.3.5 Conducting maintenance reviews

Maintenance reviews are closely matched to the Agile user feedback activity which enables the development team to pinpoint the new user needs or encountered issues. The creation of the relevant new stories and tasks is achieved during the product backlog refinement activity.

8.9.3.6 Software migration

Software migration projects can involve all or parts of the ECSS-E-ST-40 processes, from requirements engineering, up to design, implementation and validation. As such, the project can be organised following the Scrum-like SDLC, which is outlined in clause 6. One obvious benefit of doing so, is the iterative development approach, which goes along with this methodology.

One particular aspect of software migration is however, that through pure migration, typically not many new features are added to the software functionality, hence the increased involvement of the customer (which is a key motivation for adopting Agile methodology) is likely not to add much value to pure migration projects.

As a side note, it is worth noticing that migration often impacts the performance and non-functional requirements of the software product. In this case, the involvement of the customer can indeed add value, hence an Agile SDLC can be more attractive.

In particular when the software product subject to migration is composed of a number of independent components, such as DB, MMI, back-end logic, the migration plan can be aligned together with the customer to the sprint planning of the Agile SDLC.

8.9.3.7 Software retirement

There are no particular aspects of the software retirement by itself which appeal to adoption of an Agile SDLC.

Table 8-7: Mapping ECSS-E-ST-40 to Agile activities. Software Maintenance

ECSS-E-ST-40 requirement	Agile activities	Comment
5.10.2.1 Establishment of the software maintenance process	No specific Agile activity	The way to fix maintenance sprints durations and how to answer unforeseen requests is discussed with the customer and reported in the maintenance plan.
5.10.2.2 Long term maintenance for flight software	No specific Agile activity	N/A
5.10.3.1 Problem analysis	User feedback Sprint planning Product backlog refinement	N/A
5.10.4.1 Analysis and documentation of product modification	Coding, testing and documenting	While the output are produced following an Agile methodology, analysis and documentation follow rules defined in the maintenance plan.
5.10.4.2 Documentation of software product changes	Coding, testing and documenting	N/A
5.10.4.3 Invoking of software engineering processes for modification implementation	Coding, testing and documenting	Agile development guidelines defined in the preceding clauses are applicable.
5.10.5.1 Maintenance reviews	User feedback	N/A
5.10.5.2 Baseline for change	Product backlog refinement	N/A
5.10.6.1 Applicability of this Standard to software migration	N/A	N/A
5.10.6.2 Migration planning and execution	Sprint planning Product backlog refinement	N/A
5.10.6.3 Contribution to the migration plan	Sprint planning Product backlog refinement	The migration plan is discussed and agreed with the customer. It proposes an initial product migration backlog with migration stories.
5.10.6.4 Preparation for migration	Coding, testing and documenting	Agile methodology, helps in prioritising the critical areas and perform prototyping in early sprints.
5.10.6.5 Notification of transition to migrated system	No impact through adaptation of Agile methodology	N/A
5.10.6.6 Post–operation review	No impact through adaptation of Agile methodology	N/A
5.10.6.7 Maintenance and accessibility of data of former system	No impact through adaptation of Agile methodology	N/A

8.10 Independent software verification and validation

Independent Software Verification and Validation can be described a uniform, cost effective and reproducible ISVV process across projects. However, ISVV activities are often addressed with respect to the lifecycle project reviews: in the context of an Agile SDLC, it is necessary to align the ISVV process with the models identified in clause 6.6.2, although Model 3 (introducing delta reviews for changed functionality) seems to align best to the needs of a ISVV process.

Other alternatives to synchronisation with project reviews can be considered such as considering full product releases (or a number of sprints), with all the required artefacts (requirements, design, code, tests) as input for ISVV process. The timing of the provision of such full releases are carefully balanced: early releases can contain a high number of known software problem, while performing ISVV in later releases can provide modest value to the software development, due to its advanced state. Another suggested approach is to perform a two-step ISVV process with different levels of ISVV, basic for an earlier release and full for a later release of the software product.

9
Guidelines for software product assurance and configuration management

9.1 Software product assurance

9.1.1 Introduction

9.1.1.1 Main motivations for introducing Agile

Some of the main motivations for introducing Agile are:

a. the satisfaction of the customer,

b. the focus on the technical excellence. and

c. the frequent reflection about the development process.

These three Agile principles are related directly to quality. On the other hand, there is no element in the Agile manifesto or principles in contradiction with the objectives of the software product assurance discipline: "to provide adequate confidence to the customer and the supplier that the developed or procured/reused software satisfies its requirements throughout the system lifetime" (see ECSS-Q-ST-80 clause 4.1). However, Agile introduces some specific assumptions on the customer requirements and how they are identified: requirements can change during the development lifecycle and they are established through constant communication between developers, customers, users and any other stakeholder. Nonetheless there are several challenges in the deployment of a Product Assurance process in an Agile project:

- Establishing adequate responsibility and authority.
- Planning the product assurance process.
- Reporting of the product assurance activities.
- Software dependability and criticality analysis.
- Process assurance.
- Software Problem Management.
- Control of non-conformances.
- Software Development Environment aspects.

Note that verification of the different lifecycle work products is covered in clause 8.7.

9.1.1.2 Responsibility and authority

Agile promotes a more collective approach to verification of process and product compliance, therefore organisations can distribute even further the SPA function among the different team members. On the other hand, the highly incremental aspect of software development increases the importance of continuous software quality. Depending on the adopted model and the project needs, the SPA function (on customer and supplier side) could shift the focus from few major inspection points to the establishment of ongoing automated quality gates in the frame of continuous integrations. The SPA role can then proactively assure product quality with respect to product assurance requirements and quality rules that could become part of the Definition of Done. In this role, the SPA function would be in charge to collect and present to the Customer, at each iteration, evidences and guarantees of Product or Process conformity to customer requirements (e.g. that Customer-agreed documentation is available and updated and/or that the Definition of Done was confirmed for all User Stories)

There are different ways to implement the SPA role in support of increasing the confidence on meeting the requirements. For example some organisations have been experimenting with Software Product Assurance personnel trained and appointed as Scrum Masters. Since the traditional definition of the Scrum Master role is the responsible for the Agile process working and improving, resolving impediments, and communicating to external stakeholders, there is consistent logic in making this person the responsible for ensuring that the Scrum process is followed. In other instances, there can be an organisational culture in which the SPA role is already consolidated and can fit in an Agile process, for example providing an independent assessment at the iteration reviews and retrospectives.

9.1.2 Planning of software product assurance activities

Though Agile promotes responding to change over following a plan, this refers to response to scope requirements, not the set of activities to be performed. The Agile process as such should be well defined from the beginning, including the SPA activities: what kind of controls and metrics are collected, how the results are reported and analysed, how to deal with problems and non-conformities, how to handle critical software, and how improvements in the process can be deployed (see contents of Software Product Assurance Plan, ECSS-Q-ST-80 Annex B). The clause of process assurance of the plan should aim to assure that Agile values and principles are applied throughout the established process.

Like any other document in Agile, the Software Product Assurance Plan is an evolving document, adapting its components according to the needs. For example, different metrics or product controls can be necessary after a number of iterations.

9.1.3 Software product assurance reporting

The reporting of SPA activities takes a natural place at the end of each iteration and it can be twofold:

1. The sprint review where the focus can be on the Software product and the work product themselves. Though Agile principles state that “working software is the primary measure of progress” a number of metrics can be provided and collected. Most, if not all of the collection of the classical metrics should be tool-supported and automated during the development process: number of SPRs, number of passed/failed tests, code coverage, complexity, coding standards compliance, etc. Agile also introduces a number of metrics related to productivity and predictability of the process: burn-down charts (tracking the completion of work throughout the iteration), cumulative flow diagram (showing the quantity of work in a given state: under analysis, coded, verified, tested, accepted) and velocity (e.g. story points committed vs story points delivered showing the team

productivity and its ability to produce and deliver values to the customer). Those metrics can also provide an indication of which verifications have been undertaken and which ones are still to be performed.

2. The retrospectives on the other hand are the perfect occasion to assess the process and seek improvement opportunities in a continual way.

9.1.4 Technical Debt and noncompliance of Quality Requirements

The concept of quality requirements is introduced in ECSS-Q-ST-80 clauses 5.2.7 and 7.1.1 and it is based on the idea of the quality model, as a set of characteristics and the relationships between them which provide the basis for specifying quality requirements and evaluating quality. These characteristics comprise among others: functionality, reliability, maintainability… In turn they can be decomposed in subcharacteristics, for example, Maintainability is composed of: complexity, testability correctness, etc. In order to establish quality requirements these characteristics need to be defined in quantitative terms. For more information on quality requirements and quality model, please refer to ECSS-Q-HB-80-04.

On the other hand, the idea of technical debt is defined as the remediation cost for all kinds of problems in the source code. In the context of Agile, projects generate a large number of changes to the code. If the technical debt of the code for some of the quality characteristics (e.g. reliability, maintainability) is too high, then developers will be slowed down in their productivity. The more technical imperfections a software has, such as precarious architecture, poorly-written code, coding standards not followed, the more the later stages of the software development will cost. Untested code and non-covering tests are often the main origin of what is called technical debt.

Therefore, technical debt definition covers the noncompliance of the set of quality requirements that need to be dealt with before the end of the software development.

- operational non-compliance made up of functional software defects which do not meet expected behaviour, like functionality, reliability, performance characteristics,
- structural non-compliance made up of defects in term of software quality characteristics like maintainability, testability, reusability, portability, which could not lead to operational non-compliances in the first moment.

Technical debt should be assessed, using a proper metrication programme covering the applicable quality characteristics and sub-characteristics, throughout the development phase in order to monitor its evolution and gauge what would be required to rectify it. Part of the debt cannot be appreciated in the present (e.g. operational defect that has not yet been observed) and sometimes it can be difficult to be measured (e.g. it is possible that design complexity does not measure properly poor design).

As with financial debt, the more time goes by, the more that has to be paid back with cumulative interests. The highly incremental aspect of Agile development increases the importance of continuous software quality. Measuring the technical debt and keeping it under a certain acceptable level forces the development team to concentrate on the quality of the product as well as the application of processes.

9.1.5 Software criticality

The applicability of Agile methods for developing critical software has not yet been demonstrated unequivocally, and little research exists in this area. As we have seen in clause 5.2, the criticality of the software is one factor that influences the decision to use Agile. Critical software not only requires a certain amount of formality such as software project reviews with clear criteria and approvals, but analysis on how the software being developed contributes to the system dependability analysis. Then, the required verification activities are put in place and design mechanism are introduced to mitigate or reduce the software failure consequences. On top of that, formal evidence is provided that appropriate assurance activities have been performed. These constraints can result in a less Agile approach, in which requirements analysis, criticality determination, high level architecture design and overall planning are performed in a more traditional way, while incremental development with short iteration is performed in an Agile way.

However, it is necessary to assure that all activities identified for critical components are performed within the iteration. This can affect to the "definition of done", expanding it as more testing, more verification, more metrication need to be performed for critical components than noncritical. Performing these additional tasks for critical components within the iteration implies using adequate automated tools and best practices like continuous integration, pair programming and code reviews – see clause 9.1.8.

Criticality of components can be impacted by continual evolutions of the functionalities and architectural changes: if that is the case, it is necessary to re-assess the criticality at each major architectural and functional change. Technical debt management strategy for critical components, considering its most rigorous development approach, needs to be established from the beginning.

9.1.6 Software problem management

Agile provides a natural way to manage software problems, since they are identified during the previous iterations, and disposed of in the planning meetings. In some cases, problems identified are not recorded separately from the requirements and the stories in the backlog, but jointly. This in turns can provide an interesting approach since during the planning meeting, there can be a discussion regarding how to balance the introduction of new functionalities in the software product versus the value of correcting existing problems (see acceptance in clauses 6.5.5 and 8.6). The practise of adding software defects (bugs) as Product Backlog Items has the advantage of providing clear visibility of the problems in the software product, focuses the team on the need to improve SW quality and raises stakeholder awareness for managing the technical debt. In any case, the use of an automated tool is an extended practice for tracking, analysing and communicating problems both in Agile and traditional development approaches.

Some recommendations need to be considered:

- Establish the problem management process and tool from the beginning of software development. Teams are working on development functionality and testing from the first iteration. Having a place to capture problems, issues, and defects is vital for ensuring that when they have been identified, they are kept for continued focus without losing this important information.
- Keep track on which iteration the software problem has been raised. This compares with the traditional approach of identifying it in which development phase.
- When applying pair programming, it is necessary to determine the best way to adjust the responsible for the problem. In most cases the pair do not work on defects, but sometimes they do when the resolution of the defect is complex and a pair approach can be applied. Also, when

pairs have made changes based on the story tasks they have worked on, then it may be best to assign to the pair the responsibility of correcting the defect.

9.1.7 Control of non-conformances

The handling of non-conformances (NCR) on Agile is also based on the iterative approach of the process. Once requirements have been approved and passed acceptance tests and even deployed to the operational environment, yet the development iterations continue (see also different approaches for acceptance in clauses 6.5.5 and 8.6). Non-conformance Review Boards can be established at the planning meeting at the beginning of the iteration, since customer, SPA and Software Engineering representatives and other stakeholders are present and can decide how to handle the NCR, in case of correction, they can also decide in which iteration the solution should be implemented.

9.1.8 Software development environment aspects

One of the most important point to consider when deploying the SPA process in an Agile development is automation. Taking into account the relative limitation in terms of time for each iteration, and the simplicity approach that Agile advocates, then it is necessary to have tools with built-in assurance and verification activities during the lifecycle, recording the process as it happens. As we have seen: to keep the trace up to date even during frequent changes of requirements; to provide justification of the technical debt level and evidence of the quality of software product; performing code standards and code coverage checks at the time of the build; making metrics available to the decision-makers at iteration reviews and in general having up-to-date documentation and quality records accessible throughout the development cycle.

We can categorize SPA tools for Agile development as follows:

- Peer reviews are at the heart of proactive Agile SPA and need tools for the management of those. These can be done using pen and paper, but it is often supported by SW tools. It is essential to align this activity with the overall sprint logic.
- Static code checker: continuous integration software often is able to perform static code checks automatically. Such tools can check the code against agreed coding conventions, they can find bugs, identify common programming flaws, and detect copy-paste blocks.
- Dynamic code checks: continuous integration and build systems can automatically execute unit and integration tests for regression testing. They can also automatically execute acceptance and user interface tests for verification and validation purposes.
- Automated reporting: lots of automated tools can create reports like compile- and build-results, test coverage results, and several more metrics. These can be created and reported automatically; additionally it is important to define manual quality gates where quality engineers and developer review these reports and use them for continuous improvement.

9.1.9 Summary of software product assurance activities in Agile

Table 9-1 summarises the application of ECSS-Q-ST-80 requirements in an Agile software development context.

Table 9-1: Mapping ECSS-Q-ST-80 to Agile activities. Software product assurance

ECSS-Q-ST-80 requirements	Agile activities	Recommendation
5. Software product assurance programme implementation		
5.1 Organization and responsibility	Roles definition	Software product assurance engineer or manager can take also the role of Scrum master. The SPAE can also be a member of the Scrum Team, but with added responsibilities for SPA activities too. In some scenarios (typically, when model 1 or model 2 are followed), the SPAE / SPAM can have a specific role in the Project Team, independent from the SCRUM team (e.g. to support formal reviews). See clause 9.1.1.2.
5.2 Software product assurance programme management	Planning I and Planning II	Although an initial software product assurance needs to be issued at Sprint#0, this plan can be adaptive and responsive to changes in the development priorities. See clauses 9.1.2, 9.1.3, 9.1.4, 9.1.6, and 9.1.7.
5.3 Risk management and critical item control	Planning I and planning II Sprint reviews	See Risk management strategies in clause 7.2.7.
5.4 Supplier selection and control	Adopting Agile has no impact.	See concept of Scrum of Scrums, clause 7.2.4.
5.5 Procurement	Adopting Agile has no impact.	N/A
5.6 Tools and supporting environment	Sprint#0	See clauses 6.5.3 and 9.1.8.
5.7 Assessment and improvement process	Sprint retrospective	N/A
6. Software process assurance		
6.1 Software development lifecycle	Selection of an Agile lifecycle Sprint#0	Including preliminary decisions, in clause 5, definitiofn of lifecycle in clauses 6.6 and 7.3.

ECSS-Q-ST-80 requirements	Agile activities	Recommendation
6.2 Requirements applicable to all software engineering processes	See specific considerations	See clauses 9.1.5.
6.3 Requirements applicable to individual software engineering processes or activities	See respective engineer clauses	N/A
Software product quality assurance		
7.1 Product quality objectives and metrication	Sprint#0 Sprint review	Sprint#0 defines the metrication programme and the strategy of technical debt management. Sprint reviews verify the compliance with respect to quality requirements. See clauses 9.1.3 and 9.1.4
7.2 Product quality requirements	Sprint#0 Sprint review	As per previous requirement See clauses 9.1.3 and 9.1.4
7.3 Software intended for reuse	Adopting Agile has no impact.	N/A
7.4 Standard ground hardware for operational system	Adopting Agile has no impact.	N/A
7.5 Firmware	Adopting Agile has no impact.	N/A

9.2 Software configuration management

9.2.1 Introduction

There is a strong relation between Agile and Configuration Management: Agile is based on the idea of frequent change, while Configuration Management is a mechanism to manage and keep changes under control. It is clear that to implement Agile, projects need to comply with Configuration Management best practices and standards. In particular projects need to comply with ECSS-M-ST-40.

Moreover, there are a number of common Agile practices in relation to Configuration Management that have been consolidated. Most of them are in relation to the build and release aspects as final stages of the change control process.

9.2.2 Agile software configuration management challenges

Agile is best suited to situations were requirements are not stable during the development and even embraces late changes in the lifecycle. This implies a number of challenges for Configuration Management and in particular how baselines are established and changes are controlled. This can

make it necessary to look at the ECSS-M-ST-40 with a different approach, and interpret its requirements according to the specific aspects of Agile.

One of the main issues of implementing the Configuration Management in an Agile context is the establishment of configuration baselines (see ECSS-M-ST-40 clause 5.3.1.4). A different approach than the traditional baselining after each milestone needs to be taken. Agile does not follow a phased approach like traditional methods. So, baselines are neither designated nor distinguished by phases and the exit criteria of a phase. However, we suggest a number of baselines:

- Requirements and backlog baseline. As we have seen in the requirements analysis process, this is a natural baseline that is established as part of Agile iteration planning activity. This baseline is not established at the beginning of the development. It is established and evolves in an iterative manner as stories are identified and collected into the backlog. It is subject to change during each iteration planning session.
- Iteration baseline. This baseline combines the design, development, and test elements, since all three occur in an integrated manner in an iteration. This baseline includes the design, code, and test work items that get produced by the end of an iteration. This baseline continuously changes throughout an iteration where a static snapshot of the baseline occurs at the end of the iteration as a basis for discussion in the end-of-iteration review.
- Release baseline, which is the result of the work (including executables and user documents) that is delivered to the end customer. Note that the iteration baseline discussed in the previous bullet actually evolves into the product baseline. However, the iteration baseline continues to change thereafter as the team focus on the next release, while the release is defined as a static and formally recognized baseline prior to releasing to customers to ensure baseline integrity.

Change control (see ECSS-M-ST-40 clause 5.3.2) is introduced in the planning of each iteration or sprint. It is at each iteration that it is decided what needs to be changed (as well as what new features are added). Iteration planning records changes in the backlog. Change requests are recorded directly in the logs of the backlog entries. Iteration or sprint planning activities take also the role of the Change Control Board, since they identify new changes, review them, prioritize them, then commit the work.

Configuration status accounting (see ECSS-M-ST-40 clause 5.3.3 and ECSS-Q-ST-80 clause 6.2.4) refers to the provision of a product definition. It contains the records and the reports of configuration item descriptions and all departures from the baseline during design and production. In terms of software, this activity corresponds to the creation of two documents: the Software Configuration File and the Software Release document.

Taking into account in an Agile context, there can be a delivery at the end of each iteration, then the Software Release Document is provided, containing an overview of the release, the evolution since the previous version and known problems. Most of this information can be extracted automatically from the tools supporting the stories (e.g. requirements), the problem reports and the tracking of what is actually ‘done’.

The Software Configuration File (SCF) on the other hand is a much more complete configuration status accounting document. It does not only contain the inventory of the software configuration item (including source code and binary codes) but most importantly the baseline documents and the change list. The SCF is associated with its update at project milestones. However, since in Agile, the configuration change and the establishing of a new requirements baseline is produced at each iteration, it is recommended that the elements of the SCF are provided also at the end of each iteration. An option can be to join its contents with the ones of the Software Release Document.

Configuration verification (see ECSS-M-ST-40 clause 5.3.4) is the process of determining, by verification of project material and reports, that each Software Item under configuration control meets

all its technical requirements and is ready for delivery. Since in Agile requirements are not typically very stable until the third or fourth iteration (this varies from project to project), it is recommended not to start until some stability in the requirements or story baseline has been achieved, when conducting a verification of the Configuration.

Some Configuration Management tools can provide a relationship between the source code and the built executables. It is important to verify this relationship as a way to ensure that the product we release to the customer is based on the wanted source in the version control system. A big advantage of doing this, irrespective of the method, is that if the build process break, particularly when there are many changes occurring, the automated list of source files and change records from the build can help narrow down where the breakage is occurring.

A way to approach ensuring what we are building is based on what we agreed to build is to perform the CM verification at the end of each iteration. At that moment, we examine what we built based on what we said we were going to build in the iteration planning session.

9.2.3 Agile methods for configuration management

The main Agile method that relates directly to Configuration Management is Continuous integration. Continuous integration is the practice of integrating frequently individual developer's new or changed code with the common code repository. The idea is to avoid major divergences between each developer's baseline and the common repository. If the integration is performed later in the development process the discrepancies can become difficult to resolve. Some practices advocates to trigger the building process by every commit to the common repository, others back frequent integration as ten times a day.

Taking the lean approach to the maximum extension, continuous integration can be followed by Continuous Deployment, when the time between creating new code and deploying for operations by users is minimised. This is possible with a number of tools and methods for Automatic Built. Moreover it is worth noting that each step in the integration and deployment process needs to be verified according to a number of criteria. Continuous integration therefore allows to embed quality checks to be performed into the new code: not only automatic unit, integration, validation and acceptance tests can be run, but also static analysis and code rules verification are executed. Therefore providing early feedback on the code quality and suggesting immediate corrective actions when the criteria are not met.

9.2.4 Summary of software configuration activities in Agile

The following table summarises the application of ECSS-M-ST-40 requirements in an Agile software development context.

Table 9-2: Mapping ECSS-M-ST-40 to Agile activities. Software Configuration Management

ECSS-M-ST-40 requirements	Agile activities	Recommendations
5.1 General	Adopting Agile has no impact.	Software configuration requirements are applicable independently of the selection of an Agile software lifecycle.
5.2 Management and planning		
5.2.1 Configuration Management plan	Sprint#0	A definition of the strategy for configuration management needs to be defined at the beginning of the project.
5.2.2 Configuration Management interfaces	Adopting Agile has no impact.	N/A
5.3 Implementation of configuration management		
5.3.1 Configuration identification	Planning I	Baselines are defined at: requirements/backlog, iteration and release. See clause 9.2.2.
5.3.2 Configuration control	Planning I and II, sprint backlog management and refinement	Control of changes in the configuration is decided at iteration planning level, as well as backlog management and refinement activities. See clause 9.2.2.
5.3.3 Configuration status accounting	Sprint review Product release	Usually supported by specific building and reporting tools, see clauses 9.2.2 and 9.2.3.
5.3.4 Configuration verification	Sprint review Product release	Usually supported by specific building tools, see 9.2.2 and 9.2.3.
5.3.5 Audit of the configuration management system	Adopting Agile has no impact.	N/A
5.3.6 Configuration management approach for operational phase	Adopting Agile has no impact.	N/A
5.3.7 Implementation of information/documentation management	No specific Agile activity	See clause 7.6 “Software documentation plan” with regards to live and evolving documentation concept.