Geir Kjetil Hanssen
Tor Stålhane
Thor Myklebust

SafeScrum® – Agile Development of Safety-Critical Software

Springer

SafeScrum® – Agile Development of Safety-Critical Software

Geir Kjetil Hanssen • Tor Stålhane •
Thor Myklebust

SafeScrum® – Agile Development of Safety-Critical Software

Springer

Geir Kjetil Hanssen
Software Engineering, Safety and Security
SINTEF Digital
Trondheim, Norway

Tor Stålhane
NTNU
Trondheim, Norway

Thor Myklebust
Software Engineering, Safety and Security
SINTEF Digital
Trondheim, Norway

ISBN 978-3-319-99333-1 ISBN 978-3-319-99334-8 (eBook)
https://doi.org/10.1007/978-3-319-99334-8

Library of Congress Control Number: 2018954543

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This book addresses the development of safety-critical software and proposes the SafeScrum® methodology. SafeScrum® is—as the name indicates—inspired by the agile method Scrum, which is extensively used in large parts of the software industry. Scrum is, however, not intended or made for safety-critical systems, hence we have proposed guidelines and additions to make it both practically useful and compliant with the additional requirements found in mandatory safety standards. We have specifically addressed the generic IEC 61508:2010 standard, part 3 (the software part), but this book will also apply to other, related domain-specific standards. Just like Scrum, SafeScrum® is to be considered a framework and not a fully detailed process suitable for all projects. This means that each case needs to consider adaptations of the framework to make it work optimally. The ideas and descriptions in this book are based on collaboration with industry, through discussions with assessment organizations, general discussions within the research fields of safety and software, but also on the authors' own judgements and ideas. Hence, SafeScrum® and this book do not necessarily represent the view or liability of any specific organization or individual.

Safety-critical systems are increasingly based on software, while established practice is often directed towards design and development of mainly hardware-based systems. While hardware-based systems call for a high level of details early in development since hardware is costly to alter, software can be managed more flexibly throughout development. This calls for new ideas on how software should be developed efficiently and how compliance with safety standards should be managed; we believe that agile methods will offer new opportunities to a domain facing new challenges.

This book provides basic knowledge on safety-critical systems with an emphasis on software. It provides an overview of agile software development, and how it may be related to safety, and it explains how to interpret and relate to safety standards. SafeScrum® is described in detail as a useful approach to gain the benefits of agile methods and is indented as a set of ideas and a basis for adaptation and adoption in industry projects. This covers roles, processes and process artefacts, and documentation.

We look into how standard software process tools may be taken into use. We provide insights into some relevant research in this new and emerging field and also provide some real-world examples.

Trondheim, Norway
June 2018

Geir Kjetil Hanssen
Tor Stålhane
Thor Myklebust

Acknowledgements

We would like to thank the Research Council of Norway for co-funding the work leading to this book, through the SUSS research project (#228431 Smidig Utvikling av Sikkerhetskritisk Software—Agile Development of Safetycritical Software). In collaboration with the authors, Børge Haugset has contributed with developing the SafeScrum® idea, and in particular with Chap. 10. We would also like to thank our project partners, Autronica Fire & Security and Kongsberg Maritime, that have contributed considerably to the shaping of SafeScrum®. We also want to thank several assessment organizations for taking part in discussions, in particular on how to interpret the IEC 61508:2010 requirements and guidelines. The International Electrotechnical Commission (IEC) has granted us re-print of important tables and details from the IEC 61508:2010 standard. Finally—and in particular—we are grateful for the support and valuable contributions by Ingar Kulbrandstad, Frank Aakvik, Jan-Arne Eriksen, Ommund Øgaard, Erik Korssjøen and Lars Meskestad.

Contents

Chapter 1
Why and How You Should Read This Book

What This Chapter Is About

- *Why you should consider agile development.*
- *How agile development can help you.*
- *Some industrial experience.*
- *Some warning—the IEC 61508:2010 has its own use of some terms.*
- *A short summary of the rest of the book.*

1.1 The Starting Point

This book is mainly written for people who know a lot about how to make safety-critical software but little or nothing about agile development in general and Scrum[1] in particular. For example, when we discuss how Scrum improves project communication, this is a general observation and holds for all types of software development. However, it will also help when developing safety-critical software and that is why we discuss it here. Concepts related to safety are discussed for two reasons: (1) to show how standard safety analysis methods fit into an agile framework and (2) to show how safety concepts will influence agile development. For people that are already using Scrum, this book might be an introduction to safety systems development. However, it is only an introduction—learning how to analyse and develop a safety-critical system will need a lot more studying.

In this book, we present a combination of current research, as published in international, peer-reviewed journals and conferences, our experience collected during our cooperation with industry, and information found in blogs and forums. The last source might raise some eyebrows but we have seen that many interesting results related to emerging ideas and technology are published in blogs a long time

[1] Scrum is the agile method used as a basis for SafeScrum®. See chapter 2 for an introduction.

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_1

before they appear in a scientific paper. There are two reasons for this: (1) it takes some time before enough scientific evidence can be collected and (2) many practitioners are focusing on getting their job done and have little or no focus on the academic publication process.

This book is one of the results of the SUSS[2] research project—a project sponsored by the Norwegian Research Council along with two industrial partners. The main goal of the project was to adapt the Scrum development process to the IEC 61508:2010 standard, which resulted in the SafeScrum® process. The cooperation with our industrial partners helped them to use agile development and gave the project important feedback as to what worked and what did not work.

Last but certainly not least—when you start reading this book, make sure you have a copy of the IEC 61508:2010 at hand. An alternative might be the Exida book on functional safety [3].

1.2 Why Agile Software Development?

An increasing part of our society depends on software intensive systems. This software is complex and often contains safety-critical components. Examples are fire alarm systems, smart-grid, medical equipment and aviation systems.

There is a clear trend in the development of safety-critical systems that more functionality is realized through software and less through hardware. The reasons for this are increasing hardware performance, reduced hardware costs and an increasing need for flexibility and speed. Being able to rapidly apply new technologies and add new requirements is easier in software than in hardware. Standard hardware components can now be programmed, meaning less effort on hardware development and more on software development. This also means that development moves towards a higher tolerance to changes in requirements and design in the development process. Several European companies have products where nearly 100% of the development costs are related to software development. This, however, also leads to larger and more complex *software* development projects and an increasing effort needed for software assessment. IEC61508-3:2010[3] has several requirements affecting the software development process; there is however little clarity on how to do this in practice.

The common practice today when developing safety-critical software systems is to use a plan-based and document-driven development process, which leads to inflexibility in requirements change as well as large costs for producing documentation to manage the certification process. Industrial data indicates documentation-

[2] 'Smidig Utvikling av Sikkerhetskritisk Software' (Agile Development of Safety Critical Software).

[3] This notation means part 3 of the standard, in this case the part that affects the software process.

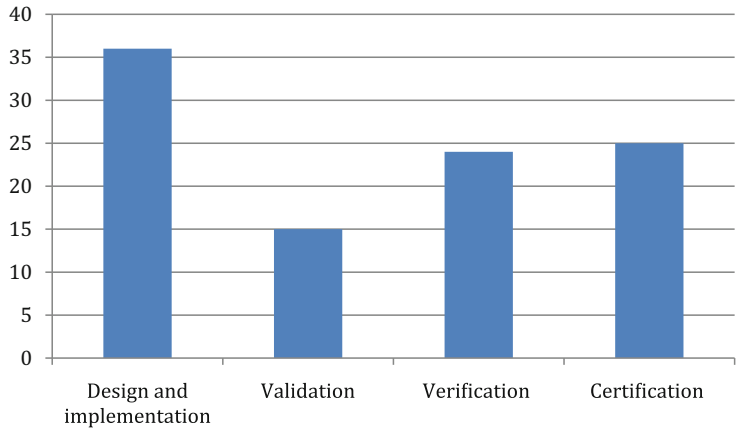

Fig. 1.1 Document-related average costs per development phases

related costs (verification and certification) between 25% and 50% of the total development costs [5]. See diagram in Fig. 1.1.

The objective of SafeScrum® is to reduce these costs by developing documents as the information becomes available instead of (1) writing the document at the start with the information that we believe to be correct and then (2) have an expensive process at the end of the project where we change the documentation according to the final facts. One of our industrial partners claimed that this problem was the main reason that they wanted to become agile.

Traditional plan-driven approaches, which are commonly used in the safety domain, do not cater to the increasing need for flexibility. We thus propose a new approach for agile development of safety-critical software systems.

1.3 Why Should the Industry Consider Agile Methods?

The industry has until now been plan-driven and methodically conservative. However, several changes in the environment have affected this:

- The speed with which new technology is introduced in the marketplace is growing; shorter time-to-market becomes an increasingly important competitive advantage.
- Increased focus on flexibility and innovation as part of the Internet-of-things trend (e.g. connected autonomous vehicles) and the growth of cyber–physical systems (e.g. wearable medical devices). This is partly a consequence of the increased speed of introducing new products and partly a consequence of the need to allow for requirement changes due to changing customer and market needs.

- There is a growing realization that the plan-driven development paradigm is much too focused on writing and rewriting plans that are not used and on producing planning documents that are not read. Agile development, with its focus on flexibility, helps us to develop documents as they are needed by management, developers, assessors or customers—not when a document plan requires it. This helps us to create documents when they are needed and with the latest available and updated information.
- The industry focuses on lean development and production [1]; whatever does not contribute to the product's final value should be removed.

There is reason to believe that the speed of inventions and innovations will increase further. The railway joint undertaking S2R (Shift to Rail), the development of autonomous vehicles, and other research and development initiatives will increase this speed and those who do not follow will quickly get into trouble. In addition, more and more developers are educated through the use of agile development outside the safety-critical domain. It remains to be seen if these programmers will want to work in a development environment based on plan- and document-driven development methods. The key benefits that come from the combination of a safety-oriented approach and a process model for agile software development are that the process enables [7]:

- Continuous feedback, both to the customer, the development team, the independent test team, and the external assessor
- Re-planning, based on the most recent understanding of the requirements and the system under development
- Mapping of functional and safety requirements
- Traceability from requirements to code and from code to tests
- Coordination of work and responsibilities between the three key roles: the development team, the customer and the assessor
- Closer cooperation with the RAMS[4] engineer and the quality assurance responsible roles
- "Test-first" development of safety-critical systems

All of these points will help us get a more visible process and thus better control over the development process, which again will help us deliver quality on time and within budget.

[4]Reliability, availability, maintainability, safety.

1.4 What Do We Have to Offer?

There are two ways to attack the challenges related to safety-critical software and agile development: we can listen to the gurus' theories or we can listen to the practitioners' experiences. We choose to follow Machiavelli and use the latter approach.

> . . .it appears to me more appropriate to follow up the real truth of the matter than the imagination of it; for many have pictured republics and principalities which in fact have never been known or seen, because how one lives is so far distant from how one ought to live, that he who neglects what is done for what ought to be done, sooner effects his ruin than his preservation; for a man who wishes to act entirely up to his professions of virtue soon meets with what destroys him among so much that is evil

This book combines two important areas: agile software development—in our case Scrum with additional practices from XP—and development of safety-critical software. The approach is general and has been evaluated with respect to several domains such as nuclear (IEC 60880:2006), railway (EN 50128:2011) and process industry (IEC 61511:2011). Our main work, however, has been in the domain of IEC 61508:2010 and especially IEC 61508-3:2010 and we have thus used this standard as the basis for our examples throughout the book. On the other hand, the principles discussed are easily adapted to the other standards—see Chap. 9. Just to add to the challenge posed by the respective standards, the process must be in accordance with the assessors' requirements. The only thing we can change at will is Scrum. Thus, we will describe how we have:

- Isolated software development from the rest of the process. This is what we call *separation of concern*, allowing software developers to focus on what they do best—develop software. What is outside the software development process but still needs to be done is named *alongside engineering*. As a consequence of this, every document that can be written before software development will be done at the start of the project. Some documents may be finished—for example, the safety plan—while others will need to be modified during the software development process. As much of this as is possible will be taken care of by the alongside engineering team.
- Adapted Scrum to the requirements of IEC 61508:2010. This is done by adding several roles—for example, quality assurance responsible—and activities—for example, traceability—to the Scrum process.
- Included other well-known and useful applicable agile practices into the development of safety-critical software, such as the daily stand-up meetings and the flexibility when it comes to requirement changes.
- Alongside safety activities—the part of alongside engineering related to safety—activities that are performed synchronized with the sprint activities but are done outside the sprints.

Some of our readers will probably not have a lot of knowledge and experience related to Scrum or to agile software development in general. We have thus included a section on this topic—see Chap. 2. Those who know agile development—especially Scrum—and want some insight into how to develop safety-critical software according to IEC 61508:2010 and to the satisfaction of a certification body should start with Chap. 3.

The IEC 61508:2010 standard is, at least in the short run, difficult to change and will mostly be taken as given. One of the authors is involved in the standard committee that updates IEC 61508:2010 and works actively to move the standard in a more goal-based way—that is, focus on the *results* instead of focusing on *how to get there*. This will allow us to focus on the development goals and then select the appropriate process, whatever that may be. This will make IEC 61508-7:2010 highly relevant since this part especially focuses on *what* to achieve.

Certification bodies can be influenced by negotiations, but only to a small degree. The main adaptions, however, will have to be done by changes and add-ons to Scrum. The leitmotif of this book is thus how to adapt Scrum so that it can be used to develop safety-critical software in compliance with IEC 61508:2010 and still get the assessors' acceptance. The result of this has been a process called SafeScrum®. In our opinion, the method is flexible and can be used for both large and small companies and projects.

In order to make sure our ideas work, we have discussed and partially tested out SafeScrum® with two large Norwegian companies producing SIL3 systems. Their feedback has been extremely important and is included into the process as we received it. We have also discussed ideas with the companies' assessors to get their point of view. To take Machiavelli's advice, we have written on something that is used, rather than some fancy ideas about what could have been done if we all were someone else in a different place at a different point in time. Enjoy!

1.5 Does It Work?

There are several case studies of the use of agile development in safety-critical systems. See Chap. 12 for a review of some of them. There is unfortunately little published experience related specifically to IEC 61508:2010. Here we will look at two cases, both on the use of agile practices—one using agile development on medical software (IEC 62304:2006) and one using it on aerospace systems (DO 178C:2012).

P.A. Rottier and V. Rodrigues, 2008 [6] The company Cochlear® started to introduce some agile practices in 2001. After some positive experiences, they decided to introduce Scrum in two development projects. The introduction of a new development process was driven from senior management. Some issues from the company's process are as follows:

- They used the user story concept. At the start, they only identified enough user stories for the first two sprints.
- They found that implementing and testing a set of user stories took more time than was allocated to each sprint. This problem was solved by introducing more automatic test tools for unit testing.
- User stories were administrated using the Confluence tool—one page per user story.
- System test was done using the Greenpepper framework, which integrated seamlessly with Confluence.

Some important experiences and lessons learned (cited from the paper):

"Although we did some up front design of the system architecture, we did not do nearly enough in terms of consciously evolving and revisiting the architecture at regular intervals.

Without those [predefined tests as in TDD (test-driven development)], tests being in place before we start on the code we have no way of reaching a finished state on any User Story.

By using test driven development along with the iterative development of features we have been able to fairly consistently produce high quality code. While we did not find the development process to be any more efficient than the process we were used to, we think this is due to the large amount of supporting frameworks we had to construct to allow it to operate in our environment. As we move forward, we fully expect to be reusing a lot of what we have developed and therefore increased efficiencies".

R.F. Paige et al., 2011 [4] The other set of experiences, which are related to the aerospace standard DO 178, focuses on testing in agile development, stating that agile development has a strong testing culture. Maintaining a comprehensive test suite allows development to proceed iteratively without letting an iteration compromise what has been achieved in the preceding iterations. Testing provides vital evidence of safety, and has a significant presence in, for example, DO-178B. In their opinion, TDD [2] is consistent with recommended practice such as DO-178B. Some adaptation of TDD may be required though to satisfy the assessor. For example, white-box testing and coverage criteria feature prominently in the standards, providing evidence that tests are adequately comprehensive. In addition, four other arguments also support agile development (cited from paper):

- *"Coding standards. These are already used extensively within high-integrity processes.*
- *Design improvement. The process of "safe" refactoring through TDD maintains a high-quality design. This is especially important if changes are inevitable, and especially when change is managed on an ad hoc basis.*
- *The planning game. The short "inspect and adjust" feedback cycle of agile processes supports dynamic project management and helps to reduce process risk. Planning data measures tangible progress from earlier in the life cycle.*
- *Emphasis on communication. "Problems with projects can invariably be traced back to somebody not talking to somebody else about something important". Agile development foster high-bandwidth verbal communication and shared responsibility, which reduces the likelihood of a single point of failure in the process".*

1.6 A Warning

Before you read the rest of this book, beware that the IEC 61508:2010 has its own definition of several terms that are also used in software development. For the complete list, see IEC 61508-4:2010, section 3—Definitions and abbreviations. Two terms which we know have already caused trouble in communication between developers and assessors are "(functional) unit" and module". Thus, for your benefit, below we show their definitions according to the standard.

- *"**3.2.3 Functional unit**: entity of hardware or software, or both, capable of accomplishing a specified purpose. NOTE: In IEV 191-01-01 the more general term "item" is used in place of functional unit. An item may sometimes include people.*
- ***3.3.5 Software module**: construct that consists of procedures and/or data declarations and that can also interact with other such constructs."*

The software module definition is elaborated in IEC 61508-3:2010 as follows:

- *"**7.4.7.2** This verification shall show whether or not each software module performs its intended function and does not perform unintended functions".*

Thus, a software module shall perform a specific function. Based on the definitions above, it is easy to see that the developers and the assessor will have dramatically different opinions when it comes to, for example, unit tests. In order to make things easy, we will use the terms "unit" and "module" the same way as software developers do. If we mean "unit" or "module" as defined by IEC 61508:2010, we will use the terms "functional unit" or "functional module".

1.7 Cooperation with Two TÜV Certification Bodies

Focus in this book is on using an agile development process for the development of safety-critical software. In many cases, the customer or the authorities will require you to have the system certified. In order to include the certifiers' concerns and requirements, we have been in close contact with TÜV Nord and TÜV Rheinland. In cases where we needed information on interpretation of the IEC 61508:2010 standard or wanted to discuss activities that could be alternatives to the standard's requirements, TÜV always provided clear and complete answers. However, the views represented here are our own and do not represent TÜV's views or policies.

1.8 What Next?

The rest of this book has the following content:

Chapter 2. What is agile software development? A short introduction.
Chapter 3. What is safety-critical software—just to get you started. We present RAMS (Reliability, Availability, Maintainability and Safety), and some thoughts on information security and some on testing.
Chapter 4. Placing agile in a safety context—what is SafeScrum® and how it should be used. In addition to the SafeScrum® introduction, this chapter also contains the important section "Preparing for SafeScrum®", describing what has to be done before you start SafeScrum® development.
Chapter 5. Standards and Certification—where we discuss the importance of standards, their relations to safety-critical software and how the standards will influence the development project. In addition, we give a short description of the quality assurance role in SafeScrum®.
Chapter 6. The SafeScrum® process—the most important part of the book. Here we describe the SafeScrum® process in details. Important issues such as sprint planning, sprint reviews, the daily stand-ups are discussed together with other important issues such as configuration management, change impact analysis and the role of the safety culture.
Chapter 7. SafeScrum® activities—provides details on how the activities in SafeScrum® can be carried out.
Chapter 8. SafeScrum® additional elements—explains important aspects such as traceability, change impact analysis, testing, safety engineering, and release management.
Chapter 9. Documentation and Proof of Compliance. Although agile development processes try to avoid excessive documentation, some documentation is still needed. This chapter tells you what is needed and why.
Chapter 10. A SafeScrum® tool chain—description of components that can be used to build an agile tool chain, which can also be used for safety-critical development.
Chapter 11. Adapting SafeScrum®. Even though this book is mainly about SafeScrum® and IEC 61508:2010, evaluations of standards have shown that we can adapt SafeScrum® to several other safety standards without any serious problems—for example, DO 178C:2012 (avionics) or EN 50128:2011 (railway). This chapter tells how it can be done.
Chapter 12. A summary of research—summing up some identified experience-based research on agile development of safety-critical software.
Chapter 13. Some examples from a real project using parts of SafeScrum®.

Annexes

A. All documentation needed to claim compliance with IEC 61508:2010
B. A short introduction to safety analysis—explaining important aspects such as FMEDA, FFA, HazId and Fault Tree Analysis

C. Useful UML diagrams
D. Analyses required by IEC 61508:2010

Glossary
An overview of some important abbreviations and acronyms.

References

1. Cawley, O., Wang, X., & Richardson, I. (2010). Lean/agile software development methodologies in regulated environments–state of the art. In *Proceedings of lean/agile software development methodologies in regulated environments–state of the art*. Helsinki: Springer.
2. Koskela, L. (2007). *Test driven: Practical tdd and acceptance tdd for java developers*. Greenwich, CT: Manning Publications.
3. Medoff, M., & Faller, R. (2014) *Functional safety: An IEC 61508 SIL 3 compliant development process.* exida. com LLC.
4. Paige, R. F., Galloway, A., Charalambous, R., Ge, X., & Brooke, P. J. (2011). High-integrity agile processes for the development of safety critical software. *International Journal of Critical Computer-Based Systems, 2*(2), 181–216.
5. Reicenbach, F. (2012). Avoiding pitfalls in safety projects – Why experiences often make the difference. In *Proceedings of ICES workshop on – Security, safety, robustness and diagnosis: Status and challenges*. Kista: Self published.
6. Rottier, P. A., & Rodrigues, V. (2008). Agile development in a medical device company. In *Agile, 2008. AGILE '08. Conference*.
7. Stålhane, T., Myklebust, T., & Hanssen, G. K. (2012). The application of Scrum IEC 61508 certifiable software. In *Proceedings of ESREL*. Helsinki, Finland.

Chapter 2
What Is Agile Software Development: A Short Introduction

What This Chapter Is About

- *We explain the central concepts of agile development as they are used in Scrum.*
- *We show the high-level, generic Scrum process diagram.*
- *We introduce some relevant agile concepts.*

2.1 Agility and Safety

Agile development methods are becoming a de facto standard for software development in nearly all domains. Documentation and plans are deliberately kept to a minimum in order to concentrate the effort on developing working software. For safety-critical systems, however, these priorities in agile software development methods like Scrum may lead to scepticism among safety engineers, who feel that agile development does not fit. The main reason for this is that these projects traditionally require that a strict plan be defined upfront. We need to keep in mind that safety-critical projects suffer from many of the same problems that mar other software development projects, such as the need to change plans and requirements, being too late and having a solid budget overrun.

2.2 Agile and Scrum in a Nutshell

For the sake of exactness, it is practical to keep the two terms "iterative development" and "incremental development" separate. Cockburn [2] has provided the following two definitions that we will use throughout this book:

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_2

- *"Incremental development is a staging and scheduling strategy in which various parts of the system are developed at different times or rates and integrated as they are completed.*
- *Iterative development is a rework scheduling strategy in which time is set aside to revise and improve parts of the system."*

Agile software development is a way of managing and organizing the development process, emphasizing direct and frequent communication, frequent deliveries of working software increments, short iterations, active customer engagement throughout the whole development life cycle, and change responsiveness rather than change avoidance. This is in contrast to waterfall-like models, which emphasize thorough and detailed planning and design upfront, and conformance to consecutive stages of the plan. Several agile methods are in use by a large part of the software industry, of which Scrum [7] and extreme programming (XP) [1] are the two most commonly used, often in combination. Scrum is project management framework, while XP is a set of practices intended to improve software quality and respond to changes in customer requirements. Figure 2.1 explains the basic concepts of an agile development model, exemplified by Scrum.

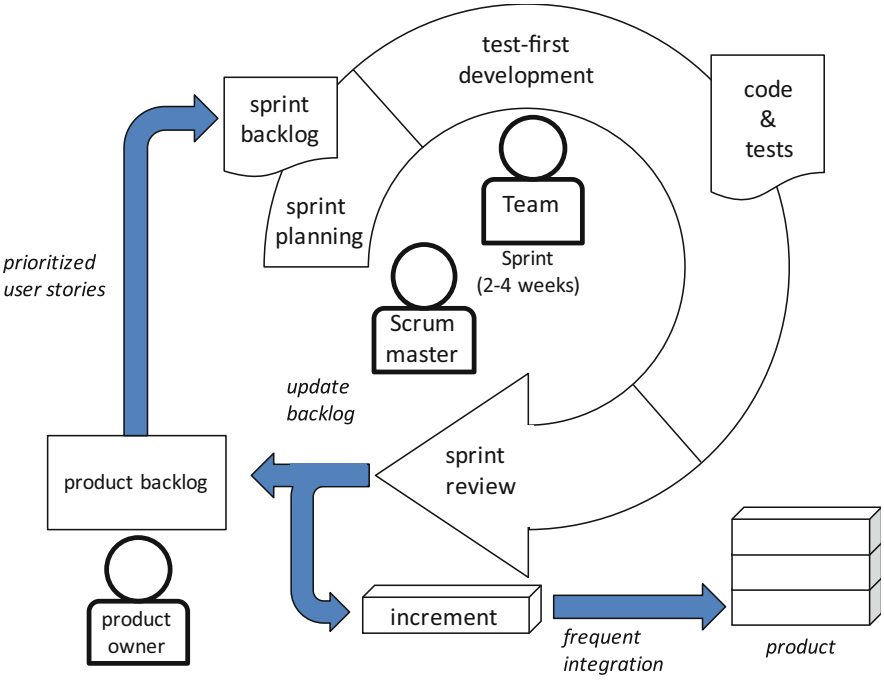

Fig. 2.1 The basic agile software development model

The description of Scrum in this chapter is deliberately kept simple and is only meant as a basic introductory for the uninitiated reader. For the interested reader, we recommend a couple of books that explain the basic principles as well as some practical experience in more detail. Henrik Kniberg's *Scrum and XP from the Trenches* [4] is a down-to-earth introduction, which is also available as a free download at https://www.infoq.com/minibooks/scrum-xp-from-the-trenches-2. Another read is Jeff and JJ Sutherlands *Scrum: the art of doing twice the work in half the time* [7].

2.3 Scrum and XP Concepts

The Scrum framework is based on a few central concepts:

- The **product backlog** is a store of jobs waiting to be done and is a concretization of the more traditional requirements specification. The product backlog consists of user stories. The product backlog is created before the first sprint, but may be updated in between sprints.
- An **epic** is a high-level description of what the user wants to achieve. It will usually be broad in scope and contain few details. In order to be implemented it needs to be broken down into two or more user stories—see next item.
- A **user story** is a short description of some functionality; its goal, its expected results, how it can be demonstrated, etc. Each user story also has a cost estimate (e.g. how many story points [3] or how many person-hours that are needed to resolve it) and a priority (how important it is to the owner of the system). Priorities may change during the project, based on growing knowledge of the system being developed and thus changing requirements (in the form of user stories).
- A **sprint** is a time boxed development period, typically 2–4 weeks, where a part of the code is developed from a set of stories. Each sprint thus builds an increment of the system and this part is integrated with the previous parts either at the end of the sprint or when the relevant tests are run and accepted—also called "continuous integration". In this way, the system (product) is built through an incremental and iterative building process. The total cost of the selected stories for a sprint needs to equal to the amount of resources available for the next sprint. The total amount of available resources (person hours) is the sum of available person-hours from the members of the development team in the next sprint and is known upfront.
- Each sprint starts with a **sprint-planning meeting** where the top priority items from the product backlog are moved to the sprint backlog—adding up to the amount of resources available for the sprint. These requirements will be implemented in the subsequent sprint. When a user story is moved to the sprint backlog, it needs to be broken down into tasks. Each task will be assigned an amount of resources. Implementing the tasks will realize the user story. For more details, see Chap. 6.

- Development is ideally based on the **test-first principle** [5], meaning that unit-tests, and often also some of the functional tests, are written prior to the code. This ensures good code design and good test coverage. In addition, it is also common to apply a framework for automated higher-level tests, typical automated acceptance testing using tools such as FitNesse or similar. The principle is the same; to frequently test new or changed code to get immediate feedback.
- The **sprint backlog** contains all stories that will be implemented in the upcoming sprint and is populated with user stories from the product backlog where the sum of estimates matches the time and resources in the sprint.
- Each sprint ends with a **sprint review meeting** where the results from the sprint are demonstrated. Stories that are found to be completed are marked as done and removed from the backlog. Stories that have an unsatisfactory result are moved back to the product backlog to be resolved later, potentially with modifications based on what has been learned from the previous sprint.
- Each working day starts with a **daily stand-up meeting**, which is a short meeting where each member of the development team explains (1) what she/he did the previous work day, (2) any impediments or problems that need to be solved and (3) planned work for the current work day.
- Each sprint releases an **increment** which is a fully functional (executable code) or, in other ways demonstrable part of the final system (presentation of DB scheme or a piece of software that runs on a simulator, etc.). New or improved code is frequently integrated with the code base.
- Alternatively, a **sprint retrospective** may be organized in between sprints to evaluate the development process itself to identify necessary process improvement actions. A retrospective may focus on three questions: (1) What went well? (2) What went wrong? (3) What should be improved? The results are used for process improvement.
- If we have a project with more than 10–12 persons, it is practical to split them up into several Scrum teams with four to six persons in each team. Each of these Scrum teams should appoint a representative who will participate in the daily **Scrum of Scrums** meetings [6]. The Scrum of Scrums meetings are run in the same way as a regular Scrum meeting and are used for coordination of multiple teams.

2.4 Scrum Roles

Scrum is based on a set of basic roles:

- The **team** are the developers and testers (and potentially other expert roles) that produce code and tests. The team should be stable over several sprints to ensure team cohesion and should together contain all competency needed to resolve the defined user stories.
- The **Scrum master** is responsible for making the Scrum process run smoothly and organizes the regular events such as the sprint planning meeting, the sprint

review meeting, daily stand-ups, and retrospectives. The Scrum master will seek to resolve any problems that may occur during the sprint. The Scrum master may also take on development tasks.
- The **product owner** is responsible for prioritizing the product backlog, and which stories that goes into the sprint backlog. Because of this, he or she is also responsible of approving the results from each sprint. The product owner directly or indirectly represents the user's interests.

2.5 Iterative and Incremental Development

The product backlog is revised by the customer and is potentially changed/reprioritized based on the importance of each backlog item and the available resources. This starts the sprint-planning meeting for the next sprint. It is important to plan the remaining activities in the project so that all "must have" requirements are met within the resource limits. Even though pure Scrum does not have "must have" requirements, a safety-critical system will have—for example, all the safety requirements must be fulfilled. When all product backlog items are resolved, the final product is released. If all the resources are spent and there is still high-priority items left in the backlog, it is up to the product owner to decide what to do—for example, add resources or reduce product functionality. The final tests—for example, a factory acceptance test (FAT)—will be run to ensure completeness and correctness. For a more in-depth description of Scrum, see Chap. 6.

References

1. Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Boston: Addison-Wesley Professional.
2. Cockburn, A. (2002). In H. J. A. Cockburn (Ed.), *Agile software development. The agile software development series*. Boston: Addison-Wesley.
3. Coelho, E., & Basu, A. (2012). Effort estimation in agile software development using story points. *International Journal of Applied Information Systems (IJAIS), 3*(7), 7–10.
4. Kniberg, H. (2015). *Scrum and XP from the trenches*. Lulu.com.
5. Koskela, L. (2007). *Test driven: Practical tdd and acceptance tdd for java developers*. Greenwich, CT: Manning Publications.
6. Schwaber, K. (2007). *The enterprise and scrum*. Redmond: Microsoft Press.
7. Sutherland, J., & Sutherland, J. J. (2014). *Scrum: The art of doing twice the work in half the time*. New York: Crown Business.

Chapter 3
What Is Safety-Critical Software?

What This Chapter Is About

- *We give a short introduction to IEC 61508:2018 and a definition of safety-critical software.*
- *We discuss briefly the challenges posed by the safety-standards relating to the development of safety-critical software—especially the RAMS characteristics.*
- *Some security issues and issues related to testing are discussed briefly.*
- *Some issues related to resilience and why resilience and agile development go together so well.*

3.1 IEC 61508:2010

The IEC 61508:2010 standard "Functional safety of electrical/electronic/programmable electronic safety related systems" consists of seven parts. From the SafeScrum® point of view, part 3 is the most interesting. However, parts 1 and 2 will also be useful. Parts 4–7 are fine as a reference.

Part 1: General requirements
Part 2: Requirements for electrical/electronic/programmable electronic safety related systems
Part 3: Software requirements
Part 4: Definitions and abbreviations
Part 5: Examples of methods for the determination of safety integrity levels
Part 6: Guidelines on the application of IEC 61508-2:2010 and IEC 61508-3:2010
Part 7: Overview of techniques and measures

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_3

3.2 On Safety-Critical Systems

IEC 61508-4:2010, which is the definition part of the standard series, does not define "safety critical software". It does, however, define safety-related systems as a *"Designated system that both:*

- *implements the required safety functions necessary to achieve or maintain a safe state for the EUC (equipment under control); and*
- *is intended to achieve, on its own or with other E/E/PE*[1] *safety-related systems and other risk reduction measures, the necessary safety integrity for the required safety functions"*

As a starting point for this book, we will define safety-critical software as follows:

"Safety-critical software is software that by failing will endanger people, equipment or the environment".

A more simple definition is that safety-critical software is software where the customer or the authorities require that the developers should use one of the standards for development of safety-critical software—more directly, "It is safety critical because we say so". Safety-critical systems are all around us. Some important examples are the software that helps a pilot fly an aeroplane, the software that runs the Automatic Cruise Controller in our car and the software used to control important hospital equipment. The effort we need to put into making software safe must be related to the consequences if it fails. We do this using a risk analysis, which will help us find the right SIL (Safety Integrity Level) value, which again will identify a set of processes, techniques and measures that need to be performed during development. For more details, see Sect. 5.4 and the software development process model described in Sect. 6.1.

Even though the standard definition of *safety-critical* seems pretty clear, there are some problems—first and foremost: where does the software's responsibility end? It is clear that software that controls a signal light on a railway is safety critical, but what about software that does not control anything but just informs the operator. Is this safety critical? There is, unfortunately not a simple answer to this question. People working with safety analysis will tell you that the answer depends on where you draw the system's limits. Is, for example, the operator part of the system?

In general, we face two challenges—see also Chap. 5:

- In order to make the system safe, we need to understand how it, by failing, can harm persons, equipment or the environment and how we can prevent this from

[1] Electrical/Electronic/Programmable Electronic.

Table 3.1 IEC 61508:2010 recommendation levels

HR	The technique or measure is highly recommended for this safety integrity level. If this technique or measure is not used, then the rationale behind not using it should be detailed with reference to Annex C during the safety planning and agreed with the assessor.
R	The technique or measure is recommended for this safety integrity level as a lower recommendation to a HR recommendation.
—	The technique or measure has no recommendation for or against being used.
NR	The technique or measure is positively not recommended for this safety integrity level. If this technique or measure is used, then the rationale behind using it should be detailed with reference to Annex C during the safety planning and agreed with the assessor.

happening. This gives rise to the safety requirement—mostly things the system should or should not do—for example, safety functions.

- Safety-critical software needs to have a high MTTF (Mean Time To Failure), especially for the safety functions. This is, however, easier said than done since there does not exist a generally agreed-upon method for developing software with a specified reliability. There is one standard (IEEE 1633:2016) that has a set of recommended practices for estimating software reliability but none of these methods has had any impact in industry and their usefulness is doubtful, especially since software reliability will depend on the user profile. The standard IEC 61508:2010 instead identifies a set of well-known techniques and measures, which are required in order to achieve the defined safety level. Each technique is graded as HR, R, - or NR, see table 3.1. Should you choose not to follow one of the requirements, you need to argue why this is not necessary, or that you will end up with the same result via other solutions. The assessor needs to agree with this view.

A supplement to a high MTTF for safety functions is fail-safe behaviour. This implies that even if the system fails, it will not harm equipment, personnel or the environment. A simple example is a robot control system that is designed to stop the robot if an error is detected in the control software (entering a safe state).

Most of the standards use a common principle: there is a prescribed method for identifying the level of safety criticality, for example, IEC 61508-5:2010. In some of the standards, once this is decided, a set of tables are presented, as for example in IEC 61508-3:2010, which define methods and techniques that are recommended or highly recommended for the software development process.

Other techniques and measures can be used. The aim for the techniques and measures is presented in IEC 61508-7:2010.

3.3 RAMS in IEC 61508:2010

RAMS is short for Reliability, Availability, Maintainability and Safety—all important characteristics for safety-critical software. The IEC 61508:2010 standard has little direct information on reliability, availability and maintainability. Software

maintenance is mentioned in part 2, sections 7.4.7 and 7.6. In addition, it has, as seen from a software developer's point of view, a rather strange statement in part 2, section 3.7.2, where it claims that "software is not maintained, it is modified". This may make sense from a hardware point of view but for software developers, software maintenance is about maintaining the system, its usefulness and its functionality, which will lead to software updates.

For a software engineer, the definition used by the IEEE 24765:2010 makes more sense since it defines maintainability as:

- *"The ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment*
- *The ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions*
- *The capability of the software product to be modified"*

There are two sections in IEC 61508-2:2010 that refer directly to maintainability:

- *"7.4.7.2: Maintainability and testability shall be considered during the design and development activities in order to facilitate implementation of these properties in the final E/E/PE safety-related systems.*
- *7.6.1: The objective of the requirements of this sub-clause is to develop procedures to ensure that the required functional safety of the E/E/PE safety-related system is maintained during operation and maintenance".*

IEC 61508-2:2010 is concerned with hardware and is thus not directly relevant to SafeScrum®. None of the standard's sections says anything about how to create a maintainable system, which is okay from a goal-oriented view. This does not, however, mean that the standard is useless for this purpose. If we follow the requirements in Tables A4 and B1 in the annexes of IEC 61508-3:2010 (shown below), there are several good advices for how to create code with high maintainability. Note that the A-tables are normative, while the B-tables are informative. However, some assessor organizations will require that both sets of tables are used (Tables 3.2 and 3.3).

Examples are: *using a modular approach*, *design and coding standards*, *no unstructured control flow and tracing requirements to design*. This confirms the SafeScrum® stance—that the techniques and measures described in the annexes of IEC 61508-3:2010 are just sound software engineering practices. Note that the choice of methods that are highly recommended (HR) will depend on the SIL—Safety Integrity Level. For a description of how to assign a SIL value to a product, see Sect. 5.4.

IEC 61508-4:2010 also refers to ISO 2382-14:1997, which is now replaced by ISO/IEC 2382:2015, for further references to maintainability and availability. Both

Table 3.2 IEC 61508:2010 Table A.4—Software design and development—detailed design

Technique/Measure		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1a	Structured methods	C.2.1	HR	HR	HR	HR
1b	Semi-formal methods	Table B.7	R	HR	HR	HR
1c	Formal design and refinement methods	B.2.2, C.2.4	—	R	R	HR
2	Computer-aided design tools	B.3.5	R	R	HR	HR
3	Defensive programming	C.2.5	—	R	HR	HR
4	Modular approach	Table B.9	HR	HR	HR	HR
5	Design and coding standards	C.2.6 Table B.1	R	HR	HR	HR
6	Structured programming	C.2.7	HR	HR	HR	HR
7	Use of trusted/verified software elements (if available)	C.2.10	R	HR	HR	HR
8	Forward traceability between the software safety requirements specification and software design	C.2.11	R	R	HR	HR

Table 3.3 IEC 61508:2010 Table B.1—Design and coding standards

Technique/Measure[a]		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Use of coding standard to reduce likelihood of errors	C.2.6.2	HR	HR	HR	HR
2	No dynamic objects	C.2.6.3	R	HR	HR	HR
3a	No dynamic variables	C.2.6.3	—	R	HR	HR
3b	Online checking of the installation of dynamic variables	C.2.6.4	—	R	HR	HR
4	Limited use of interrupts	C.2.6.5	R	R	HR	HR
5	Limited use of pointers	C.2.6.6	—	R	HR	HR
6	Limited use of recursion	C.2.6.7	—	R	HR	HR
7	No unstructured control flow in programs in higher-level languages	C.2.6.2	R	HR	HR	HR
8	No automatic type conversion	C.2.6.2	R	HR	HR	HR

[a]Some table entries in IEC 61508:2010 are marked with a number followed by a letter—e.g., 3a and 3b in Table 3.3. Table entries with the same number but different letters are considered alternatives. Thus, in Table 3.3 you have to choose either 3a or 3b.

of these standards are titled "Information Technology—Vocabulary" and are solely concerned with definitions. The RAMS-relevant definition is as follows:

> *"Reliability, Maintainability and Availability: ability of a functional unit to be in a state to perform a required function under given conditions at a given instant of time or over a given time interval, assuming that the required external resources are provided".*

Thus, the IEC 61508:2010 contains no direct or indirect advice on how to achieve maintainability, reliability or availability—which all are important for achieving a

Table 3.4 Safety integrity levels—target failure measures for a safety function operating in high demand mode of operation or continuous mode of operation (IEC 61508-1:2010, Table 3)

Safety integrity level (SIL)	Average frequency of a dangerous failure of the safety function [h-1] (PFH)
4	$\geq 10^{-9}$ to $<10^{-8}$
3	$\geq 10^{-8}$ to $<10^{-7}$
2	$\geq 10^{-7}$ to $<10^{-6}$
1	$\geq 10^{-6}$ to $<10^{-5}$

Table 3.5 Target failure measures for a safety function operating in low demand mode of operation

Safety Integrity Level (SIL)	Probability of failure on demand of the safety function (PFD_{avg})	Risk reduction factor
4	$\geq 10^{-5}$ to $<10^{-4}$	100,000 to 10,000
3	$\geq 10^{-4}$ to $<10^{-3}$	10,000 to 1000
2	$\geq 10^{-3}$ to $<10^{-2}$	1000 to 100
1	$\geq 10^{-2}$ to $<10^{-1}$	100 to 10

high level of service for the system. What is worse, however, is that it has no requirements for these characteristics. Even so, the tables in Annexes A and B in IEC 61508-3:2010 provide several good advices for writing maintainable code.

The IEC 61508:2010 does say something about reliability, albeit in a rather condensed form. The following table summarizes the relationship between the SIL value and MTTF. Bear in mind that these numbers at the present have a low confidence and are only used for risk computation purposes.

Table 3.4 shows the probability of a dangerous failure for a safety function per hour—PFH. Note that there is approximately 10^4 hours in a year. Thus, a failure rate of 10^{-5} per hour means one failure in 10 years per system. Mind you, if we have 100 systems installed and in operation, this means 10 failures per year.

For a safety instrumented system (SIS) we need to be able to assess the probability of failure on demand. A SIS is used to keep a process under control and consists of a set of sensors used to observe the system's state, a logic resolver and one or more control elements used to change the system's state. Thus, the SIS' purpose is to lower the system's risk. Table 3.5 shows how a SIS, developed to a predefined SIL will influence the number of failures per demand. The probability of failure on demand—PFD—is the probability that the SIS will fail when it is required. Thus, a PFD of 10^{-2} means that the function will fail in 1 out of 100 times it is called on, or in other words—the risk is reduced by a factor of 100.

If we introduce the parameter MTTR (Mean Time To Repair), we can define availability—the A in RAMS—as follows:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

For all practical situations, the MTTF is much larger than MTTR and we can thus write

$$\text{Availability} = 1 - \frac{\text{MTTR}}{\text{MTTF}}$$

Thus, in order to achieve high availability, we need to construct a system so that it:

- Does not lose state information based on already acknowledged input
- Is easy to restart and easy to get back to the relevant state

For a software-intensive system, MTTR is the average time from when the system goes down or stops functioning, till the system is up and running again in the state it was when it stopped and with all information restored.

3.4 Security

While the safety discipline has long traditions, learning and evolved standards, the security domain is a new one. Safety is concerned with protecting an environment from the system, while security is about protecting the system from its environment. A system most often has one or more access points, and security measures are used to prevent unauthorized access or undesired manipulation.

Security issues are identified during phase 3[2] (Hazard and risk analysis) and phase 4 (Overall safety requirements), and results in requirements like physical locks, passwords on computers, encrypted devices and air gap (physically separating the system network from the world). While access could previously be hindered through physical means, this now requires secure solutions for logically restricting access through solutions like virtual private networks (VPN).

In an ideal world, we should fix security problems straight away, in order to prevent somebody from exploiting them. In the real world, however, it is not that easy. Changes due to safety-critical problems may require extensive revalidation, testing and assessment. For these reasons, security fixes are often postponed until the next release, whenever that is. An alternative solution to the immediate fix of security breaches could be the following:

- Early in the development project—before the architecture is fixed, include "Security breaches" as part of the general hazard analysis
- Decide on alternative ways to handle this situation, for example, taking the system off the network or disabling parts of the system to be able to mitigate or encapsulate the security breach.
- Decide how to make the architecture support your alternatives for handling security breaches.

[2]Referring to the IEC 61508:2010 safety life cycle. See part 3, chapter 1 in the standard.

The simple process described above will help you think through your security challenges and make early, important decision on how to handle them.

Once this is accepted as secure enough, the system is also deemed safe and could be deployed. The problem with this approach is that while safety measures are rather static in nature, security threats keep changing—there is an ongoing arms race between security measures and ways to crack them. This is further complicated by changes in the nature of safety-critical systems. For example, there is now an ongoing attempt to bridge the safety-critical parts of airplanes and the entertainment systems on board—reducing weight by cutting down on cables by sharing common infrastructure. The air gap will instead be replaced by technical, logical separation through solutions like VPN (virtual private network). Across domains, safety-critical systems start to use publicly accessible networks, and rely on security solutions. To make these systems safe, the security solutions should be continuously watched. Any arising issues should then be evaluated—how does this affect the safety of the system? When does the risk of patching the system outweigh the risk of not doing so?

Security is one of six significant technical changes in the new edition of IEC 61511-1:2016 compared to the previous edition. While IEC 61508:2010 is a standard for manufacturing systems, the IEC 61511:2016 is a standard for designers, integrators and users of safety instrument systems. The topics 1, 2 and 3 have been kept from the first edition while the three topics 4, 5 and 6 are new requirements.

The six security requirement topics are as follows:

1. Maintenance/engineering interface.
2. Enabling and disabling the read-write access.
3. Forcing of inputs and outputs in PE SIS (Programmable Electronic Safety Instrumented System).
4. A security risk assessment shall be carried out.
5. The design of the SIS shall be such that it provides the necessary resilience against the identified security risks.
6. Information shall be contained in the application program or related documentation:

   (a) That the correctness of field data is ensured.
   (b) That the correctness of data sent over a communication link is ensured.
   (c) That communications are made secure.

In addition, the new edition provides more guidance on security, and includes reference to the following standard and guides:

- IEC 62443-2-1:2010, *Industrial communication networks—Network and system security—Part 2-1: Establishing an industrial automation and control system security program*
- ISO/IEC 27001:2013, *Information technology—Security techniques—Information security management systems—Requirements*
- ISA TR 84.00.09:2013, *Security Countermeasures Related to Safety Instrumented Systems(SIS)*

3.5 Testing

Testing is an important part of any software development project and even more so for a project developing safety-critical systems. However, it is even more important when we use an agile process. The reason for this is that in an agile process, code is changed more often—for example, due to refactoring—and we need to test that the changes due to refactoring or requirements changes do not destroy the parts of the system that are already working.

First, a word of warning: testing is not meant to replace code review. Neither will code reviews replace testing. While testing will focus on dynamic relationships, the code review will mostly focus on static relationships. Both are needed and both are important if we want to develop high-quality code. Note that there exist several static code analyses tools, which may be used.

Broadly speaking, there are two types of testing—the testing done by the developers—inside testing—and the testing done by others. Inside testing is unit testing and integration testing, done to check that each code unit satisfies its requirements and to see that the units can work together as intended. Outside testing is done by other personnel due to the standard's requirements for independence—see Sect. 6.3 and IEC 61508-1:2010 8.2.16–8.2.19. This includes RAMS testing, system testing, FAT and SAT. These tests are important for showing that the system works as intended—that is, fulfils the SRS (Systems Requirements Specification) and delivers the required services to the customer. Thus, the results from these tests—for example, test logs—are an important part of the arguments in the safety case—see Sect. 8.4.3—to show that all requirements are fulfilled.

We will give a short description of each of the testing processes in the following text. For all types of tests, it is practical to save them, together with their results and a description of which system or system part they test. In this way, they can be reused and function as a safety net if we need to change the code. The test results might also be needed as proof of compliance with the standard.

Unit Testing
Unit tests are written, run and analysed by the developer for one code unit (a procedure, class or method). Their main purpose is to check that the unit fulfil its requirements. Since they only concern a small part of the system, it is necessary to write stubs and drivers to get the tests to run. There are, however, tools that support unit testing.

Module Testing
There are several definitions of the term software module. We will use a simple one—a single block of code that can be invoked in the way that we invoke a procedure, function, or method. As for unit testing, the purpose of module testing is to check that the module provides the required services as expected. Since a module will consist of several integrated units, the module test is a stop on the way from unit testing to integration testing.

Integration Testing
Integration testing is done to see if two or more units or modules cooperate as intended. Just as unit tests, they only concern a part of the system and it is necessary to write stubs and drivers to get the tests to run.

RAMS Testing
RAMS testing is done outside the SafeScrum® process but we have a safety engineer role that connects SafeScrum® and RAMS—see also alongside engineering (Sects. 6.1 and 6.3). The safety engineer's responsibility is to connect information from the SafeScrum® team needed in the RAMS assessment and to give the necessary feedback to the SafeScrum® team. The purpose of the RAMS process is to check reliability, availability, maintainability and safety. The focus is, however, mostly on safety. One of the purposes of the RAMS testing is to test the safety functions. When asking for clarification on this topic, TÜV Nord answered as follows: *"According to EN IEC 61508 it is relevant that an independent person make tests of the relevant safety functions. It must be not a person from outside of the company. The automatic tests can be done by the same person, code review and system tests please from an independent person"*.

For more on independence related to testing, see Sect. 6.3 and IEC 61508-1:2010, parts 8.2.16–8.2.19.

System Testing
System testing is performed on the complete system—that is, all functionality is implemented. The test focuses on checking that all requirements are implemented and perform as expected. For this reason, we recommend that the system tests be written together with the requirements and that testers, developers and customer or a customer proxy all are involved in this process. The test may be run on the real hardware or in a simulator. Usually, actuators and sensors are simulated in order to be able to test a wide range of situations.

Although it might have been possible to test some non-functional requirements earlier, system testing is our first opportunity to test such things as response time.

FAT (Factory Acceptance Test)
The FAT is usually the same as the system test but run on real hardware to test responses in real situations. The results from these tests are the most important input to the safety case.

SAT (Site Acceptance Test)
The SAT will be run on the customer's hardware and on his premises. In most cases, only the FAT will be run but the customer will be free to test it in any way he or she sees fit as long as they stay inside the agreed-upon requirements.

3.6 Safety and Resilience

3.6.1 What Is Resilience?

This book is about how to develop software that is fit for safety-critical applications, mostly according to IEC 61508:2010. However, there are two types of threats to safety and they have to be handled differently:

- Known threats. Here we can use our arsenal of safety analysis methods, for example, HazId, FMEA and FTA, as mentioned in the Annexes of part 2 and part 3 of IEC 61508:2010 as well as in Annex B in this book. Using one or more of these methods will give us a set of safety requirements, which we then implement during the project.
- Unknown threats related to safety and security. Since we do not know what they are, they cannot be analysed and thus not defended against. This is where resilience comes in.

In a world where the customers' needs and operating conditions and environment changes more quickly than before, we will frequently be confronted with new threats, relating to both safety and security even though our focus is on safety. Consequently, we will need to change our systems or the way they are operated frequently.

Resilience is "the ability of a system to handle unexpected situations and recover". It is not the errors situation that is unknown but the time, place and manifestation. For example, we do not know where and under which circumstances a system hang might occur but we can still program watchdogs to discover the time-out and implement a process that will take the system to a safe state. We should perform resilience engineering with a focus on how to detect early and if possible, avoid, handle—ideally without disruption (but may be by going to a reduced state) and then recover—that is, fail and move to a safe state as quickly as possible and then get back on track and learn through agile, iterative learning.

There are three areas that need to be handled if we want to achieve resilience: a resilient software development process, a resilient software system and resilient organizations, both for development and for operation. Since resilience is such a large area, all of these are needed. There are several patterns that can be used to realize the resilience requirements but each will add work to the project while they are not needed according to the standard. Thus, there must be an identified need. This requires that we have an idea of what will be needed in the future—how the environment will change and what new challenges we will be confronted with.

Several strategies can be used. The most important ones are:

- Manage margins close to performance boundaries.
- Build common mental models based on common interfaces and terminology.
- Redundancy—have several independent ways of performing a function.
- Reduction of complexity—going from proximity to segregation, from common mode connection to dedicated connections.

- Reduction of couplings—enabling processing delays, flexibility in order of sequencing and flexibility in methods used to reach a goal.
- Graceful degradation, preferably followed by adaptation—controlled degradation and the ability to "rebound or recover", when system functions or barriers are failing. This is the most important resilience strategy.

The common mental models are related to the development process. It *"encodes the programmer's expertise and background knowledge"*—[3]. Thus, a common mental model is important if developers are going to cooperate efficiently. The rest of the strategies listed above are related to software requirements. Over time, the environment and the threat-picture will change so the software will also need to undergo changes. As a result, high maintainability will also contribute to a resilient system.

3.6.2 A Resilient Development Process

A paper by Dove [2] discusses the technical and human perspectives of agile security. It states that agile projects create proactive innovation with *"speculative assemblies for unknown needs"*. In addition, the self-organization found in agile development is a good model for "self-organizing resilient responses". A similar view is found in a paper by Black et al. [1], where they claim that *"more software will be adaptive, changing itself to cope with new requirements or unforeseen circumstances or to ensure resilience in harsh environments"*.

The agile development process is good for achieving resilience for two reasons: (1) it is easy to add new requirements and (2) the process has two forums for discussing changing requirements—the daily stand-up and the sprint reviews. The daily stand-ups will give us the opportunity to speculate over yet unknown needs, while the sprint review, which also involves the product owner, will be an opportunity to discuss new system needs forced upon us by changes in the environment. In this way, we will be able to catch future changes and take action before a crisis develops. Note that this is an important activity and necessary time has to be allotted.

As opposed to IEC 61511:2016, IEC 61508:2010 does not mention resilience. However, a lot of what is needed to develop resilient software is already included in the process if we use IEC 61508:2010. If we follow the requirements of Annex A—Table A.2, we will cover management of margins (item 3b), redundancy (item 3e) and graceful degradation (item 4b). If we include Table A4, we will also have reduction of complexity and coupling. The problem is that several of these requirements are not required for all SIL-value. Redundancy is only required for SIL 4 and graceful degradation is only required for SIL 3 and 4.

The only issue that is not taken care of is common mental models. However, since this is about project communication, it should—at least in theory—be taken care of by the agile development process.

3.6.3 A Resilient Organization

There are really two organizations involved—the development (project) organization and the operating organization, and resilience will be an issue for both. This holds especially for the need to build a common mental model. To keep a system resilient after the first delivery, the two organizations need to communicate, since the operating company will experience new problems as needs and environments change. Thus, we need an open channel where operational experiences can be fed back to the developers so that they can improve the software and maybe also their processes.

In addition, the operating company needs to become resilient in the sense that the people using the software must be able to handle situations where the system is no longer providing the specified or expected services. Graceful degradation is part of resilience but it is important that the personnel are able to handle whatever the system cannot do under the current circumstances. This will again put requirements on operators' training. In addition, the experiences from such events will go back into the requirements for the next version of the system. Thus, DevOps[3] will be an important part of a resilient system.

References

1. Black, S., Boca, P. P., Bowen, J. P., Gorman, J., & Hinchey, M. (2009). Formal versus agile: Survival of the fittest. *Computer, 42*(9), 37–45.
2. Dove, R. (2010). *Pattern qualifications and examples of next-generation agile system-security strategies*. In Security Technology (ICCST), 2010 I.E. International Carnahan Conference. IEEE.
3. Storey, M. A. D., Fracchia, F. D., & Müller, H. A. (1999). Cognitive design elements to support the construciton of a mental model during software exploration. *Journal of Systems and Software, 44*, 171–185.

[3]Unified development and operations of a system.

Chapter 4
Placing Agile in a Safety Context

What This Chapter Is About

- *We explain the important ideas in SafeScrum®—separation of concern and the relevant parts of the V-model.*
- *We present the overall SafeScrum® process—Scrum plus the needed add-ons.*
- *First we discuss important issues such as prioritizing activities, issues related to the development of safety-critical software.*
- *Then we discuss issues related to safety culture, IEC 61508:2010 information items and how to prepare for SafeScrum®.*

4.1 The Big Picture

A lot of the material we will discuss in this chapter is regular Scrum activities and not specific for SafeScrum®. We have still included it, however, just to give the reader the full picture of the process.

Our variant of Scrum, SafeScrum®, is motivated by the need to make it possible to use methods that are flexible with respect to planning, documentation and specification, while still being acceptable to IEC 61508-3:2010, as well as making Scrum a practical and useful approach for developing safety-critical systems. In order to achieve this, we have (1) separated software development from the rest of the IEC 61508:2010 process (see Fig. 4.1), and (2) extended Scrum with important activities such as two-way traceability (see Fig. 8.1). All risk and safety analyses on the system level are done outside the sprints as part of the alongside safety engineering (our term for safety activities that are relevant to, but not part of SafeScrum®), including the analysis needed to specify the target level of safety integrity (SIL).

The Scrum development process is related to the more traditional V-model. This is shown in Fig. 4.2. Note that software design is both inside and outside of Scrum. The reason for this is that design is a two-step process: high-level design, which

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_4

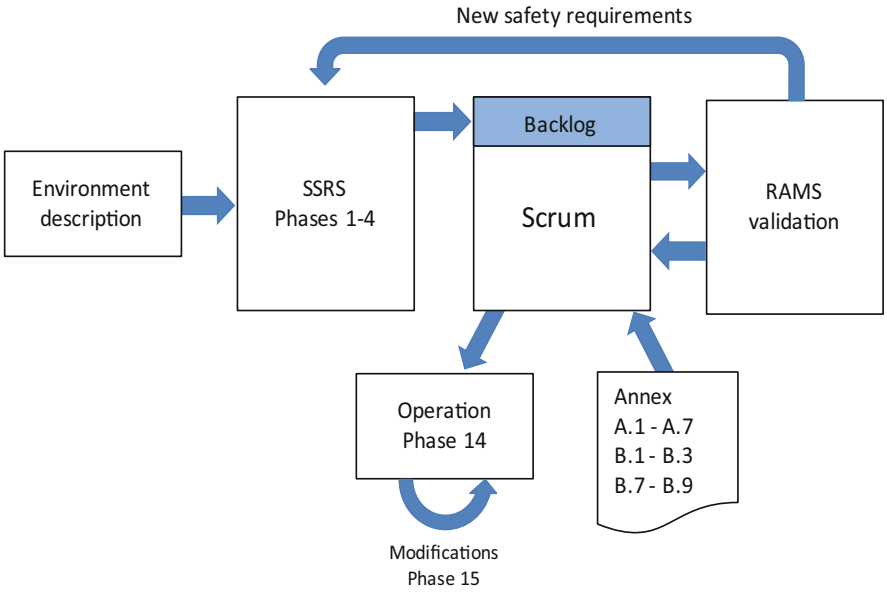

Fig. 4.1 SafeScrum® and separation of concerns

usually is done outside Scrum, and detailed design, which is inside Scrum. For simple systems, the design process inside Scrum will usually be sufficient. For SafeScrum® we have decided to focus on software system design and module design (also called detailed design). See also IEEE 24765:2010.

- *"Software system design is the process of defining the software components, modules, interfaces and data for a software system to satisfy specified requirements.*
- *Module design is the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented."*

Just as the design process, the safety analysis and risk assessment is a two-step affair. We will do as much as possible of both safety analysis and risk assessment outside SafeScrum®, but as (1) our understanding of the customer's needs grows, (2) the requirements change or (3) we make new decisions related to the realization of the requirements, we may need to repeat the analyses.

The core of the Scrum process is the *iterations*—sprints in the Scrum terminology. Each sprint consists of planning, development, testing, and verification—and is thus a mini-project on its own. An overview of the SafeScrum® development process is shown in Fig. 4.3. SafeScrum's® additions/extensions to the regular Scrum process are marked as call-outs.

A detailed description of all the roles involved in Fig. 4.3 can be found in Chap. 6.

All risk and safety analyses on the system level are done outside the SafeScrum® process, including the analysis needed to specify the target level of safety integrity

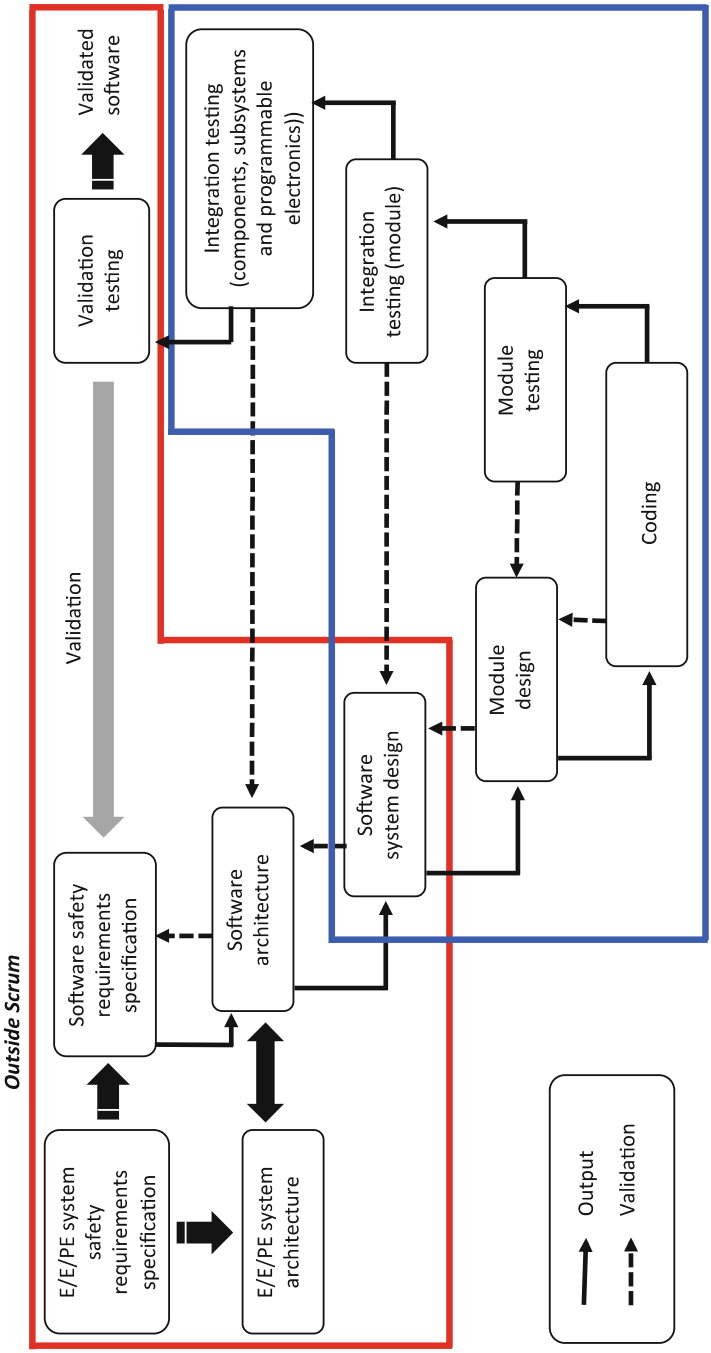

Fig. 4.2 Scrum's role in the V-model as shown in IEC 61508-3:2010

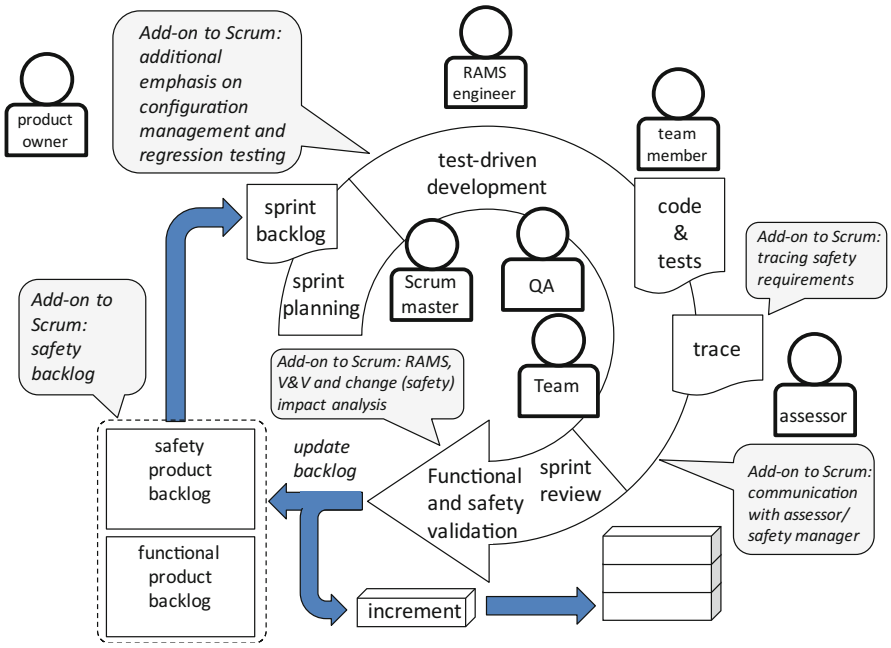

Fig. 4.3 The SafeScrum® model

(SIL). However, since the world change and our understanding of the operating environment and the system increases over time, it is beneficial to repeat parts of the safety analysis as part of each sprint-planning meeting. Which parts of the safety analysis that should be repeated will depend on the circumstances. If we change some code, the trace information will indicate what should be re-analysed. If we add a new function or change an existing one, we will as a minimum have to repeat functional safety analysis—for example, functional FMEA.

Software is considered during the initial risk analysis. Safety-related software issues should also be considered during each daily stand-up and at the sprint reviews to keep safety at the forefront of everybody's mind. Just as for testing, safety analysis also improves when it is done iteratively and for small increments. One important point is to involve the assessor early, and present the proposed method for development, for example, as part of the safety plan. The assessor will have his own views on how for instance documentation should take place, and uncovering discrepancies related to this before actual development takes place is much cheaper than resolving it later. If the project has not yet appointed a safety assessor, a safety expert could be used in this role.

Due to the focus on safety requirements, we propose to use two project backlogs: one *functional project backlog*, which is typical for Scrum projects, and one *safety project backlog*, which is used to handle the safety requirements. Adding a second backlog is an extension of the original Scrum process and is needed to separate the frequently changed functional requirements from the more stable safety requirements. With two backlogs, we can keep track of how each item in the functional

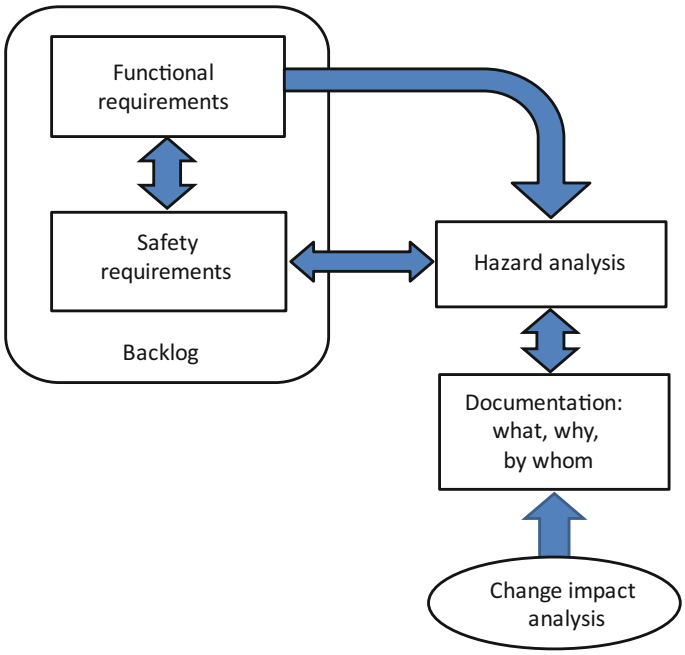

Fig. 4.4 Safety requirements versus functional requirements in the backlog

product backlog relates to the items in the safety product backlog, that is, which safety requirements are affected by which functional requirements. This can be done by cross-referencing the two backlogs and can also be supported with an explanation of how the requirements are related, if this is needed to fully understand a requirement. The separation does not need to be physical—it can be accomplished by using different tags in a common backlog—see Fig. 4.4.

Figure 4.4 also shows how the change impact analysis is related to functional requirements and safety requirements. The change impact analysis starts with the documentation of what should be changed, why it should be changed and who will do the job. When this is specified, we perform a hazard analysis based on the relevant requirements—functional and safety. Note that safety requirements and functional requirements in the backlog may be coupled. For example, a safety requirement is inserted in order to keep a specific function behave in a safe manner.

We have three types of safety requirements for software:

- **Process requirements** come from the applied standard—for example, IEC 61508:2010—Annexes A and B. These requirements are mostly related to the process and will not be placed in a backlog. They will, however, influence the way we develop the software, for example, by describing special activities during analysis, design or testing. Thus, process requirements have been taken into consideration in the description of SafeScrum®.
- **Barrier requirements** are ***not*** part of a function but are a separate piece of the system software that is used to handle a dangerous situation. A typical example is

a procedure to check the signal sent to an actuator. These requirements should go into the safety part of the backlog.

- **Software requirements** are part of a software function or method. A typical example is a piece of software that handles a set of sensors where the software is required to raise an exception if the sensor readings are outside a predefined range. These requirements are safety parts of a functional requirement and could be part of the functional requirement or separate safety requirement linked to the functional requirement.

The decision whether to, for example, insert code for control of data inside the function itself or as a separate barrier function will vary between projects. A useful rule could be that small checks—for example, a range check, is inserted into the function while a check involving more information—for example, the system's state—should be written as a separate barrier function.

Experience from one of our industrial partners indicates that we should use only one backlog if the majority of the requirements are safety requirements. In this case, all requirements should be treated as safety requirements. The main reason for this is that having two versions of the process—one for the non-safety-critical part and the same process with add-on to cater to the safety requirements—may lead to confusion for the developers.

For the development of safety-critical systems, we need two-way traceability between program code and requirements from their origin, both for functional requirements and for safety requirements. The documentation and maintenance of the tracing information is introduced as a separate activity in each sprint, preferably automated as much as possible. This activity generates the trace documentation—see Chap. 6. In order to be performed in an efficient manner, traceability requires the use of supporting tools. Several process-support tools can manage this type of traceability in addition to several other process support functions.

An iteration in Scrum starts with the selection of the top prioritized items from the product backlog—see Sect. 4.2. In the case of SafeScrum®, items in the functional product backlog may refer to items in the safety product backlog, thus creating requirement interdependencies. The staffing of the *development team* and the duration of the sprint (1–4 weeks is common), together with the estimates for each item decide which items to select for development. The selected items constitute the *sprint backlog*, which ideally should not be changed during the sprint (this should be done *between* sprints). In the development phase of the sprint, the developers produce code to address the items selected from the sprint backlog.

A sprint should always produce an increment, which is a piece of the final system. During development, this should be executable code, but it may also be user interface mock-ups, database designs, documentation, FMEA/FMEDA, code analysis, etc. The sprint ends by demonstrating and validating the outcome, to assess whether it satisfies the items in the sprint backlog, as seen from the product owner's viewpoint. Some items may be found to be completed and can be checked out (marked as *done*), while others may need further refinement in a later sprint and thus have to go back into the backlog. To make Scrum conform to IEC 61508:2010, the

final validation in each iteration consists of both a validation of the functional requirements and safety analysis to address safety issues. Depending on the SIL and the consequences of failures, personnel that are not participating in the project must be involved if a functional unit will be tested. Increments that are not complete functional units are tested by the SafeScrum® team alone. If appropriate, the persons responsible for V&V[1]—the RAMS engineer and the product owner—may take part in the validation of each sprint. The final V&V may have been done by an independent team—see IEC 61508-1:2010, part 8.2.16–8.2.19.

There are issues that require involvement and coordination of several actors. One example of such behaviour is that code cannot be added to the master branch in the repository until someone other than the developer has reviewed it, and this decision has been reviewed and approved in plenum by the whole team, the product owner and the RAMS engineer. Those responsible for V&V should also take part in the review after each sprint to help the team to keep focusing on safety considerations. If confusion or deviation from the relevant standards arises, the assessor should be contacted for discussions as quickly as possible. Using an iterative and incremental approach means that the development project can be re-planned during each sprint planning meeting, based on the most recent experience with the growing product. Between the iterations, it is the duty of the customer or product owner to use the most recent experience to re-prioritize the product backlogs. This supports an important principle in agile methods; learn continuously and adapt immediately to avoid late discovery of problems.

Applying the RAMS validation process at the end of each increment will also give the initial risk and safety analyses a gradually evolving scope. Safety analysis performed on small increments will be more focused and thus give better results. This will improve the quality of the safety analyses. Even if the increments cannot be installed at the customer's site, they can still be tested and run as part of a system simulation. For SafeScrum®, a dedicated *RAMS engineer* role has been added.

As the final step, when all the sprints are completed, a final RAMS validation will be done. Most of the developed system has been incrementally validated during the sprints. Thus, each component in the system has been analysed and tested. In addition, the components have been integrated with the other components available at that point in time. For this reason, the final RAMS validation is expected to be less extensive than when using other development paradigms—for example, a waterfall project with one, final, big integration. This will also help us reduce the time and cost needed for certification.

[1] Verification and validation

4.2 Prioritizing

We will discuss two approaches to assigning priority to the backlog items—effort/value-based and test/convenience-based—also known as "shift left"—see [3]. In practice, any priorities may be overruled by the RAMS engineer. The reason for this is mostly that he or she wants to test or evaluate some safety mechanism early in the development process. In some sense, this can be considered as a "shift left" decision—see the end of this chapter.

The effort/value diagram is shown in Fig. 4.5. At first glance, it is straightforward but experience shows that it is difficult for the product owner to say that anything is of low value. At the start of the process, everything is claimed to be important. On the other hand, the developers will usually have a fairly good idea of the effort needed. To handle the value assessment, we suggest that you create a set of scores to be used for this.

The shift left prioritizing mechanism has as its goal to enable system tests as early as possible. The reason for this is that the most serious errors are not coding errors—which are discovered using unit tests—but requirements and design errors, which can only be discovered by functional testing.

In order to be able to do functional testing, we need to pick user stories that will realize a complete function. This will prioritize the user stories. Functional areas are often divided into several epics and numerous user stories, involving several application components/modules—see Chap. 2.3—Scrum Concepts. Thus, as part of the

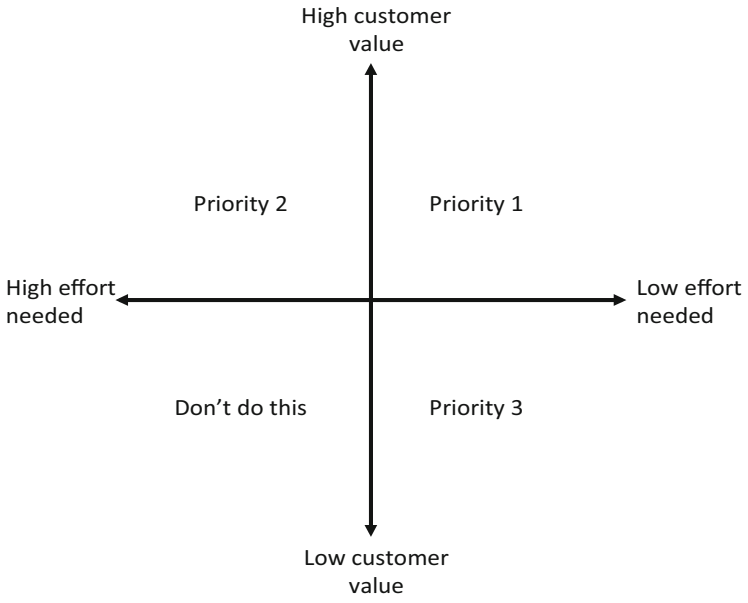

Fig. 4.5 The effort–value diagram

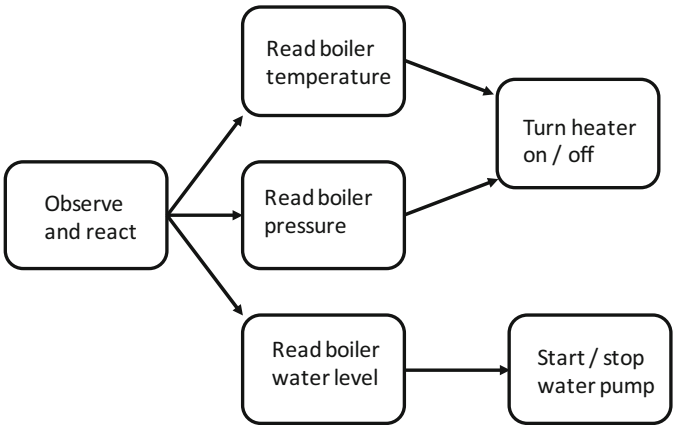

Fig. 4.6 Example of user story dependencies

sprint planning, the Scrum team must create user story dependency maps. These are used to make sure that the development is planned and completed in a correct sequence with the purpose of completing development in a timely fashion so that system testing of complete areas could start as early as possible. The dependency map also includes a map of module dependencies for each user story. In addition, the dependency maps are used in status and progress reporting as well as in informal cross-team communication.

Fig. 4.6 shows a simple example of a possible set of user story dependencies. The goal (epic) is to control temperature, pressure and water level in a steam boiler.

If we want to run functional tests early, we could for instance prioritize the controller itself (Observe and react), plus reading the water level and starting or stopping the water pump. On the other hand, it would be less helpful to start by implementing the three read-functions or, for example, the start/stop the water pump.

4.3 Development of Safety-Critical Software

We will take the development process from IEC 61508-3:2010, as our starting point. At the top level, the process model for development of safety-critical software is the same as for any other software development—analysis followed by realization (implementation), operation and maintenance.

When we move from the top-level view—analysis, realization and maintenance—to the next level of details, the process can be described as regular software implementation but with some important add-ons:

- We start by getting an overview over how the system in operation can harm people, equipment or environment. After this, we need to specify how the identified hazards can be removed or mitigated. This will give us the safety

requirements. The necessary steps are covered in the safety analysis. We need to do hazard and risk analysis—phase 3, identification of overall safety requirements—phase 4, and overall safety requirements allocation—phase 5. The last activity is important since that is when we decide how each safety concern is catered to—software, hardware, mechanisms, operational procedures or operator training.

- In addition to the implementation, the realization includes overall safety validation planning—phase 7—together with plans for operation and maintenance—phase 6—and installation and commissioning—phase 8. Many non-safety-critical projects also include such activities—especially plans for installation and maintenance, but for safety-critical systems, these activities are mandatory.
- Issues related to operation and maintenance has gotten too little attention in general software development. However, both overall installation and commissioning—phase 12—are important for safety-critical software—for example, in the offshore business. The other activities in operation and maintenance—validation and maintenance—are also found in general software development, albeit often with other labels. This part of a software life cycle is outside the scope of SafeScrum®. This will become more important when we also include DevOps into the development process.

4.4 The Role of Safety Culture

4.4.1 Introduction

We have borrowed the following definition of culture from "Business in Context" by D. Needle [14]:

> *"Organizational culture represents the collective values, beliefs and principles of organizational members and is a product of such factors as history, product, market, technology, and strategy, type of employees, management style, and national culture. Culture includes the organization's vision, values, norms, systems, symbols, language, assumptions, beliefs, and habits".*

There are a few keywords that are important here:

- Collective values and norms—values and norms shared by the team or organization.
- Beliefs, principles and habits—how we do things here. This is not a set of rules and regulations but a behaviour that is taken for granted in the organization.

4.4.2 What Is a Safety Culture

We have done a thorough search and found that all that has been published on safety culture is concerned with a company culture used to avoid accidents in the workplace, for example, the factory floor or on an offshore platform. The approach we need is different—how to make a culture that supports the development of safe products. A short, and to the point, definition of a company with a safety culture is a company that will rather not release a product which they think is unsafe—safety first. This implies that the developers have the necessary:

- Competence on safety, including solid domain knowledge. This is needed in order to:
  - Understand the consequences of a failure.
  - Be able to suggest barriers so that failure will be handled before they have serious consequences.
- Confidence in their own judgement—we know what we are talking about.
- Empowerment—they should not be overruled by management. The first time the management says that a timely release is more important than safety, the safety culture goes down the drain, never to be seen again.

For an agile company with a strong safety culture, safety must be on top of the agenda from day one and be an issue in all daily stand-up meetings plus the sprint review. This can be achieved, for example, by having issues related to safety and safety concerns as a fixed point on the agenda. In this way, we make sure that safety always is on top of everybody's mind. Safety is not an add-on to be applied at the end of a project. It helps if the developers know some basic methods for safety analysis, such as HazId, FMEA, Functional Failure Analysis and Fault Tree Analysis—see Annex B for more information. Even if we have a first version of the safety analysis done before the development starts, we may need to change it during development. Knowing some safety analysis methods will enable the developers to do part of the analysis needed during development if something changes. This will be of special importance in an agile project, where decisions frequently can be changed due to new requirements or new understanding. Experience with safety analysis will also build understanding of the need for safety and thus the safety culture.

Another way to look at safety culture in general is the socio-technical model of Grote and Kunzler [8], which is shown in Fig. 4.7. The three important components are proactiveness, socio-technical integration and value-consciousness. The process of creating a safety culture starts with the proactive integration of safety into the organizational structures together with values and beliefs that will prompt integration. This is followed by integration of technology and organization with norms related to automation and beliefs. During this process, it is important to take into account both values, beliefs and the relevant norms.

As most other models of safety culture, this model focuses on integration of safety in all structures and processes, optimization of technology ***and*** work organization,

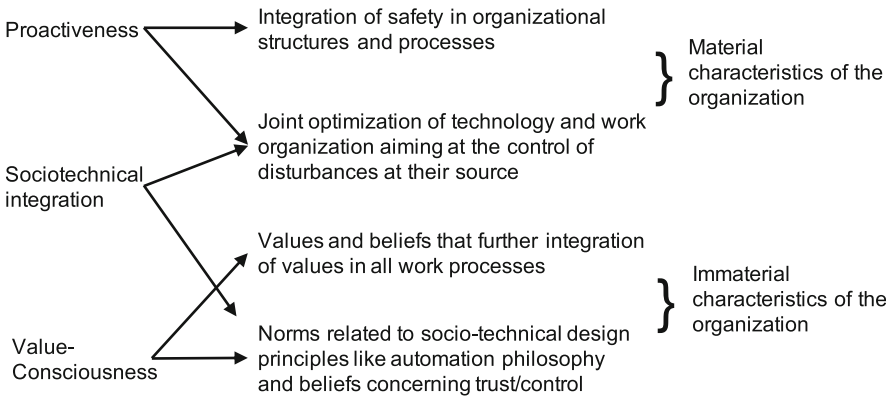

Fig. 4.7 Socio-technical model of safety culture

the values and beliefs and the established norms. For application of this model, see the following section.

4.4.3 How to Build and Sustain a Safety Culture

Based on the earlier discussion, we need to do the following to create a safety culture (listed in prioritized order):

- It must start with a management decision—safety comes first—followed up by management commitment. This implies that the management accepts, for example, that releases are delayed due to safety problems or that extra resources are needed in a project due to safety concerns or safety problems. This, however, should not prevent management from stopping a project—for example, "We cannot use enough resources on this product to make it safe enough, thus we terminate the project".
- The management decision must be followed by developer or team empowerment. The Business Dictionary [1] defines empowerment as

> *"A management practice of sharing information, rewards, and power with employees so that they can take initiative and make decisions to solve problems and improve service and performance. Empowerment is based on the idea that giving employees skills, resources, authority, opportunity, motivation, as well as holding them responsible and accountable for outcomes of their actions, will contribute to their competence and satisfaction".*

It is easy to see that the idea of empowerment fits well with agile development in general and especially with SafeScrum®. To quote Moe et al. [13]: *"The Scrum team members are empowered and expected to make day-to-day decisions within the project. They are also expected to always select the task with the highest priority when commencing work on items in the sprint backlog"*.

- The developers need to understand safety—both risk assessment and safety analysis. This is needed so that they can do their own analysis instead of being dependent on somebody else telling them that something is safety critical. The alongside engineering team will do the heavy stuff—the upfront hazard analysis and safety requirements—but the developers should be able to do simple analysis and understand the analysis done by the experts. This is important since the analysis results will influence their work. These safety analyses come in addition to the activities described in the relevant standards—for example, IEC 61508-3:2010, appendices A and B—it is not a replacement of these activities.
- The developers need to understand the application domain. It is impossible to understand a safety analysis or use it during development without this understanding. This is always important in agile development and will be even more important for agile safety development where the team is self-sustained and thus responsible for development and decisions. However, the RAMS engineer will provide important assistance when requested by the SafeScrum® team.

If we want to apply the model of Grot and Kunzler [8] to create a safety culture when developing safety-critical software using SafeScrum®, we need to make sure that agile development also affects the organization. To quote Sommer et al. [17], *"Methods that do not change the company culture will have little or no effect on important parameters such as quality and cost-effectiveness"*. Values and beliefs related to safety are strengthened by always keeping safety on the agenda—for example, during the daily stand-ups and sprint reviews. In our opinion, trust and control is a sine qua non for agile development in general and thus also for SafeScrum®. All this put together will create a safety culture.

4.4.4 A Site Safety Index

The following list from Exida is a good starting point if you want to assess the safety culture at your site. As a result of several field failure studies done by Exida over many years, we have strong evidence that failure rates for the same product vary from site to site. The ratio ranged from 2X to 4X based on product type. The differences seem to be related to site training, site procedures and other variables that we here have called safety culture. Exida defines this variable in a four-level model called the Site Safety Index (SSI) [4]. Table 4.1 was made for electronic and mechanical systems but it is a simple task to use it also for software. The parts that are relevant for software are written in italics and bold.

Table 4.1 Site Safety Index (SSI) model

Level	Description
SSI 4	**Perfect**—***Repairs are always correctly performed. Testing is always done correctly and on schedule***, equipment is always replaced before end of useful life, equipment is always selected according to the specified environmental limits and process compatible materials, electrical power supplies are clean of transients and isolated, pneumatic supplies and hydraulic fluids are always kept clean, etc. This level is generally considered to be extremely hard to achieve, but possible in some organizations.
SSI 3	**Almost Perfect**—***Repairs are correctly performed. Testing is done correctly and on schedule,*** equipment is normally selected based on the specified environmental limits and a good analysis of the process chemistry and compatible materials. Electrical power supplies are normally clean of transients and isolated, pneumatic supplies and hydraulic fluids are mostly kept clean, etc. Equipment is replaced before end of useful life, etc.
SSI 2	**Good**—***Repairs are usually correctly performed. Testing is done correctly and mostly on schedule***, most equipment is replaced before end of useful life, etc.
SSI 1	**Medium**—***Many repairs are correctly performed. Testing is done and mostly on schedule,*** some equipment is replaced before end of useful life, etc.
SSI 0	**None**—***Repairs are not always done. Testing is not done***, equipment is not replaced until failure, etc.

SSI is a quantitative model that allows the impact from what many people call "systematic failures" to be realistically included in SIL verification. SSI can provide a way to show the cost impact of alternative operational and maintenance processes.

Agile development in general and especially SafeScrum® will help to build and improve the company's safety culture. There are several reasons for this:

- Agile development teams create a transparent working environment. The daily stand-ups, sprint reviews and retrospectives make sure that everyone knows what everybody else is doing.
- Agile development focuses on correcting problems as soon as possible. If the development is done according to SafeScrum®, there is also a test-first process in place, thus making sure that all corrections are correct—see SSI 4 in the Site Safety Index.
- Adding the safety issue to the daily stand-ups and also to the sprint review will help to keep safety issues at the top of the agenda for everyone, thus first creating and later maintaining the safety culture.

4.5 Information Items

The items (documents or other forms of information) in the list below contain information that has to be made available during development of safety-critical software. This information is needed by the assessor in order to evaluate whether the development process is in accordance with the requirements in the standard, for the given safety integrity level. We will provide a quick walk-through and relate the

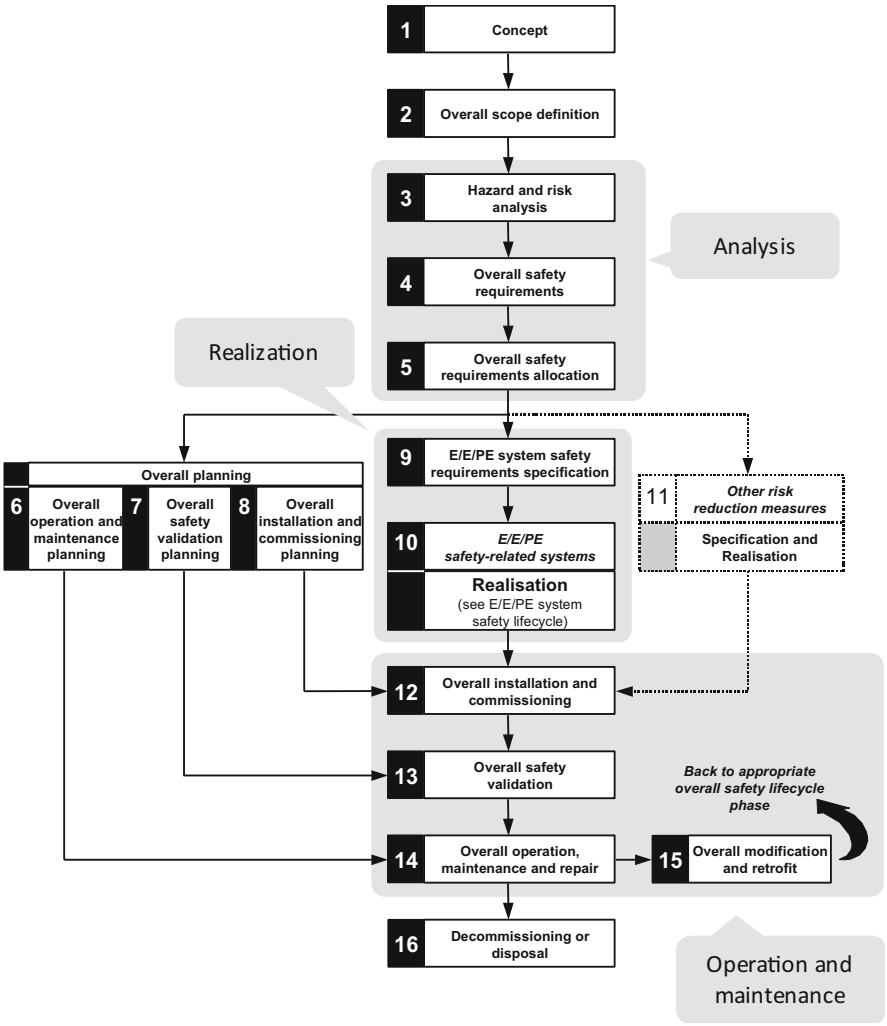

Fig. 4.8 The development process according to IEC 61508:2010

information to the development process in Fig. 4.8 and to the SafeScrum® process, shown in Fig. 4.3—Sect. 4.1. For a complete and authoritative definition of the terms in the list below, the reader should consult the IEC 61508:2010 standard. For some of the terms, the IEC 61508:2010 standard does not contain a definition, just a description of its purpose and content. In these cases, we have used the IEEE standard glossary—IEEE 24765:2010, the IEEE Standard for safety Plans—IEEE 1228:1994 or P1012/D18:2016—a standard in the making for systems and software verification and validation. Note that the text below only defines the terms. It is

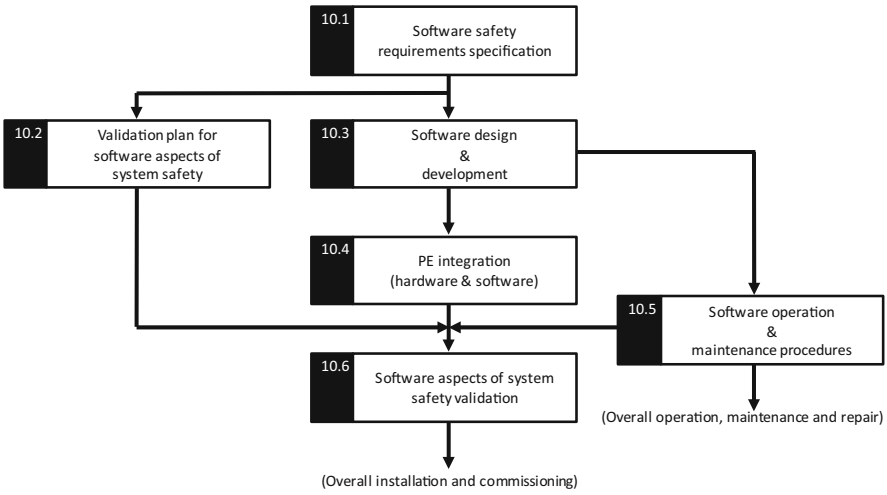

Fig. 4.9 Relationships between some of the documents defined below

important also to check the relevant standards for a ***description of the purpose*** of each document (Fig. 4.9).

It is important to note that ISO 9000:2015, which is a general quality assurance standard, has changed its definition of "document" as follows:

> *"Examples of documents are record, specifications, procedure document, drawing, report, and standards. The medium can be paper, magnetic storage, electronic or optical computer disc, photograph or master sample, or combination thereof".*

The standards put no conditions on the format, language or formalization used, other than those related to a chosen method—for example, message—sequence diagrams or state machines. Thus, we can choose whatever we think is appropriate. However, our choice should be agreed with the assessor early in the project.

The following list shows information items that are affected by the SafeScrum® process, including a reference to the part of the standard series where they are found:

System Safety Requirements (IEC 61508-1:2010)—The objective of the requirements is to define the system safety requirements, in terms of the system safety function requirements and the system safety integrity requirements, in order to achieve the required functional safety. The system safety requirements specification shall be derived from the allocation of safety requirements and from those requirements specified during functional safety planning. This information shall be made available to the safety-related system developers. The system safety requirements specification shall contain requirements for the safety functions and their associated safety integrity levels—see Chap. 5.4. Note that system safety requirements contain requirements to both hardware and software.

Software Safety Requirements (IEC 61508-3:2010)—The objectives of software safety requirements are to specify the requirements for the following:

- Safety-related software in terms of the requirements for software safety functions and the requirements for software systematic capability—see next bullet point
- The software safety functions for each E/E/PE safety-related system necessary to implement the required safety functions
- Software systematic capability for each E/E/PE safety-related system necessary to achieve the safety integrity level specified for each safety function allocated to that E/E/PE safety-related system

  The software safety requirements are found in the document "Software safety requirements specification"—see below.

Software Systematic Capability (IEC 61508-1:2010, sections 8.2.16 and 8.2.17)—defines the required independencies between developers and those who perform safety validation. The standard has a set of requirements that shall be used to decide whether the testers can be in the same project, in the same organization or belonging to another organization. This decision will depend on the product's SIL and the seriousness of failure consequences. As always, it will not hurt to include the assessor also in this decision.

Safety Requirements (IEC 61508-1:2010)—a set of all necessary overall safety functions shall be developed based on the hazardous events derived from the hazard and risk analysis. This shall constitute the specification for the overall safety function requirements. The overall safety functions to be performed will not at this stage be specified in technology-specific terms since the method and technology of implementation of the overall safety functions will not be known until later. During the allocation of overall safety requirements, the description of the safety functions may need to be modified, to reflect the specific method of implementation. For each overall safety function, a target safety integrity requirement (required SIL value—see Chap. 5.4) shall be determined. Each requirement may be determined in a quantitative and/or qualitative manner. This shall constitute the specification for the overall safety integrity requirements.

Perceived Safety Needs The safety needs as understood by the customer. This is not necessarily the same as safety requirements. IEC 61508-3:2010 describe this as *"Minimise complexity and functionality: review to ensure that all software safety requirements are actually needed to address system safety requirements"*.

Software Safety Requirements Specification (IEC 61508-1:2010)—Specification of software safety requirements, comprising software safety function requirements and software safety integrity requirements.

Software Architecture (IEC 61508-3:2010)—*"The software architecture defines the major elements and subsystems of the software, how they are interconnected, and how the required attributes, particularly safety integrity, will be achieved. It also defines the overall behaviour of the software, and how software elements interface and interact. Examples of major software elements include operating*

systems, databases, EUC input/output subsystems, communication subsystems, application program(s), programming and diagnostic tools, etc."

Software Design (IEC 61508-3:2010)—"*Software (system) design: the partitioning of the major elements in the architecture into a system of software modules; individual software module design; and coding. In small applications, software (system) design and architectural design may be combined. The nature of detailed design and development will vary with the nature of the software development activities and the software architecture. In some contexts of application programming, for example ladder logic and function blocks, detailed design can be considered as configuring rather than programming*".

Software Design Specification IEEE P1012/D18:2016—Draft Standard for System, Software and Hardware Verification and Validation—contains a representation of software created to facilitate analysis, planning, implementation, and decision-making. The software design specification is used as a medium for communicating software design information and may be thought of as a blueprint or model of the system.

Module and Integration Test Specifications (IEC 61508-3:2010)

- Module test—Testing that the software module correctly satisfies its test specification is a verification activity. It is the combination of code review and software module testing that provides assurance that a software module satisfies its associated specification, that is, it is verified.
- Integration test—The software system integration test specification shall state the division of the software into manageable integration sets; test cases and test data; types of tests to be performed; test environment, tools, configuration and programs; test criteria on which the completion of the test will be judged and procedures for corrective action on failure of test.

  See also the warning in Sect. 4.1.

The System and Software Design Requirements for Hardware/Software Integration (IEC 61508-3:2010)—Integration tests shall be specified during the design and development phase to ensure the compatibility of the hardware and software in the safety-related programmable electronics. Close cooperation with the developer of the E/E/PE system may be required in order to develop the integration tests. The software/PE integration test specification (hardware and software) shall state the split of the system into integration levels; test cases and test data; types of tests to be performed; test environment including tools, support software and configuration description and test criteria on which the completion of the test will be judged.

Software Safety Validation Plan (P1012/D18)—A plan for evaluating the safety of a system or components during or at the end of the development process to determine whether it satisfies the specified safety requirements.

Software Modification Plan (P1012/D18)—Software modifications may be derived from requirements specified to correct software errors (e.g. corrective); to adapt to a changed operating environment (e.g. adaptive); or to respond to additional user requests or enhancements (e.g. perfective). Modifications of the software

system shall be treated as development processes and shall be verified and validated as such.

Software Verification (Including Data Verification) Plan (P1012/D18)—A plan for evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Note that conditions here are the standard's requirements to the component and the development process.

Overall Operation and Maintenance Planning Write a plan for operating and maintaining the safety-critical systems, to ensure that safety is maintained during operation and maintenance. If a subsystem is taken off-line for testing, the system safety shall be maintained by additional measures and constraints. The safety integrity provided by the additional measures and constraints shall be at least equal to the safety integrity provided by the system during normal operation. IEC 61508:2010 does not consider the use of DevOps [11]. However, if at all possible, the operation and maintenance plan should include how the DevOps process should fit into the plan. See also IEC 61508-1:2010, section 7.7—Overall operation and maintenance planning.

Overall Safety Validation Planning Write a plan for the safety validation of the safety-critical system. The plan shall, among other things, include:

- Specification of the relevant modes of the operation with their relationship to the safety-critical system
- The technical strategy for the validation (analytical methods, statistical tests, etc.).
- The required environment in which the validation activities are to take place, the pass and fail criteria and the procedures for evaluating the results of the validation, particularly failures

See also IEC 61508-1:2010, section 7.8—Overall safety validation planning.

Release Plan Planning software releases is an important part of the overall planning process. Due to the needs in agile development we will split the releases into two parts—internal and external. Before a software release, the software baseline shall be recorded and kept traceable under configuration management control so that it is possible to reproduce each software release—both external and internal.

- Internal releases are aimed at developers for testing and analysis. Software is integrated and released for testing as soon as it is uploaded to the integration servers. We should re-run previous integration tests, FATs and SATs and new tests for the change.
- External releases are meant for the customers and may only be released after proper testing, analysis and certification of the safety-critical functions. There are two types of external releases:
  - Major user releases based on a stabilized development
  - Minor releases used to address minor bugs, security issues or critical defects

A release plan with fixed dates upfront will promise the customer to deliver a certain set of functionality at a certain time. This can be achieved by controlling the story priorities—whatever shall be in the next release must have the highest priorities. As a consequence of this, the release plan and the story priorities must be coordinated if one of them is changed.

Regression testing must be included in the plan for each release. Regression testing is needed to show that (1) the latest changes did not introduce an error in already existing functionality and (2) the changes did not re-introduce already fixed errors.

External releases shall come with a release note which shall include information on

- All restrictions in using the software. Such restrictions are derived from, for example, non-compliances with standards, or lack of fulfilment of all requirements.
- The application conditions, which shall be adhered to.
- Compatibility among software components and between software and hardware.

Overall Installation and Commissioning Planning We need to write a plan for (1) the installation of the safety-critical systems in a manner that ensures that the required safety is achieved and (2) that the commissioning of the safety-critical systems is done in a manner that ensures that the required functional safety is achieved. The installation plan for the safety-critical systems shall, among other things, specify

- The procedures for the installation
- The sequence in which the elements are integrated
- The criteria for declaring all or parts of the E/E/PE safety-related systems ready for installation and for declaring installation activities complete

The commissioning plan of the safety-critical system shall, among other things, specify the procedures for the commissioning and the relationships to the steps in the installation. The overall installation and commissioning plan shall be documented. See also IEC 61508-1:2010, section 7.9—Overall installation and commissioning planning.

Note that the first versions of the documents summed up in Sect. 4.5 are made before the SafeScrum® process starts—in the planning and analyses phases. Some of them may even be reused "as is" from earlier projects. However, the documents can and should be updated during the development process when we get new information, more experience or changed requirements. Agile development will make sure that this is done in a timely and efficient manner—when it is needed, and to the extent deemed necessary. Reassessment might be needed, even if the documents are reused "as is". If the documents are changed, they must be assessed as new documents. The effort needed will, however, depend on the type and amount of changes.

Development Plan This plan shall have four main points: descriptions of

- What we are going to develop

- What is the development process going to look like—especially the distribution of work between the SafeScrum® team and the alongside engineering team
- Any special requirements for the SafeScrum® team or the alongside engineering team, for example, competencies
- How we will handle interactions with management

Assessor Plan The assessor plan is mostly about how and when the assessor will interact with the development team and the purchaser of the system. The independent safety assessor (ISA) or assessor team is an independent agent hired to carry out the safety assessment. The IEC 61508:2010 mentions the assessor role several times but gives no further guidance to the assessor plan. See also Chap. 6.3 for requirements for assessor independence. We have based this chapter on the railway standard EN 50129.

The assessor's task according to EN 50129 is to *"... determine whether the design authority and the validator have achieved a product that meets the specified requirements and to form a judgement as to whether the product is fit for its intended purpose"*.

An independent safety assessment will normally consist of following up safety and quality assurance activities, and pointing out matters on the way that need to be improved. The work will result in reports with conclusion, recommendations concerning the approval processes and conditions for use. The systems and objects to be examined will be restricted to those portions that involve safety functions.

- **Kick-off meeting**—the start of the contact with the assessor. Important topics at such a meeting are, among other things:
  - Schedule issues like the development plan and assessor plan, relevant regulations and standards.
  - Schedule document scrutiny. Documents are scrutinized by the assessor as they are finished. The main documents to be scrutinized are the safety plan and the safety case, including their references. The degree of rigour depends on the safety risk to be assessed and which documents are assessed. It is up to the assessor how each document shall be scrutinized—some are scrutinized quite carefully, some are skimmed, while for some they just look at the table of contents and the conclusion.
  - Deliveries of documentation to the ISA, including how they shall be delivered.
  - Access to documents and information like databases and tools to be used.
  - Technical issues of interest for the product or system that shall be developed and assessed.
  - How findings by the assessor shall be reported is often discussed at these meetings since there are several methods. A common method is to use list of open points, including questions, clarification needed, recommendations, non-conformities, etc.
- **Safety meetings, technical meetings and RAMS meetings**—These meetings depend on the project size, contracts between the involved parties and the

complexity of the product or system to be assessed. In large projects there are often regular RAMS, technical or safety meetings. The assessor takes part in some or all of these meetings. The safety meetings normally focus on plain safety, while the technical meetings discuss special technical issues like, for example, weather protection or SIL allocation. The RAMS meetings include the RAM aspects together with safety and it is sometimes also a RAMSS meeting, which also involves the information security aspect. The meetings normally focus on:

  - Quality and safety management system
  - Technical aspects of the product and system
  - Competence and experience

- **Document scrutiny**—The scrutiny process is linked directly to the development activities and those of the alongside engineering team. However, in an agile setting, there is always a risk that some changes will occur later. If a document is changed later, the assessor can choose to only scrutinize the new parts. In all cases, the assessor will scrutinize the safety case and all documents referred to in the safety case.
- **Safety audits**—An assessor plan must schedule the necessary safety audits. These systematic and independent examinations performed to determine whether the procedures specific to the requirements of a product comply with the planned arrangements, are implemented in an efficient way and are suitable to achieve the specified objectives. These audits should document compliance with the required standard—in our case IEC 61508:2010. The main deliverable of the safety audit is the safety assessment report (SAR), which is based on the safety case.
- **Independent safety assessor deliverables**—The assessor shall deliver a list of open points and the safety audit report. An audit report should be as short as possible. Only the findings are of interest after the audit. Normally, one page is sufficient for the general information. In addition, the audit findings should be described. Transportstyrelsen in Sweden has published a letter titled "Requirement on the content of an assessment report" which is simple, pragmatic and easy to follow, when the safety case was being developed according to EN 50129. The main point is that the safety assessment report shall have a one-to-one link to the safety case, thus having the same chapter headings as the safety case.

The assessor may also, through its guidance, point out possible faults or shortcomings of a product or system. It is, however, the manufacturer's responsibility to find the technical solutions. The assessor may be invited to one or more sprint reviews. In addition, the development team and the alongside engineering team should get acceptance from the assessor if they plan to do something outside the standard. Note that the assessor cannot be asked to solve a problem and thus be kept hostage to a decision. He or she can, however, accept or reject a suggested solution to a concrete problem.

4.6 Preparing for SafeScrum®

4.6.1 What Should Be Done

First and foremost, we need to discuss how to introduce SafeScrum®. Once we have that in place, there are some decisions that need to be made early in the project, mainly because they will influence everything that comes later. Important issues are:

- Choice of language for architectural design and detailed design. We need to get the architectural description as early as possible in order to get an early start on hazard analysis. We also need a language to make and discuss design decisions during each sprint. More on this in Sect. 4.6.3.
- Choice of coding standard and metrics used to control the development process. Coding standards need to be enforced from day one.
- Method(s) for configuration management (CM). Configuration management must be in place before we start to write and change code and text.

In addition to these important issues, we will also include a short discussion on how to combine agile development and a stage-gate model.

4.6.2 Introducing SafeScrum®

First, let us reflect a little over what Machiavelli has to say about change:

"And it ought to be remembered that there is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction of a new order of things. Because the innovator has for enemies all those who have done well under the old conditions, and lukewarm defenders in those who may do well under the new. This coolness arises partly from fear of the opponents, who have the laws on their side, and partly from the incredulity of men, who do not readily believe in new things until they have had a long experience of them".

A process like SafeScrum® may represent a radical shift to many well-established organizations developing safety-critical systems. This industry tends to be quite conservative, relying on the V-model or variants of this, with heavy investment in upfront planning, prior to implementation. This affects both how the company is organized and how processes are managed. Introducing SafeScrum® may thus be challenging.

Based on action research in Norwegian industry, we have gained some practical insights on *how* such as process introduction should be done.

Adaptation and Adoption of a Radically Different Process Needs Change Agents The change to SafeScrum® could be supported by external researchers or others with updated knowledge on agile methods and on safety-oriented development and the IEC 61508:2010 standard series. However, the detailed shaping of SafeScrum® and the change itself has to be driven by a small and motivated team of developers that act as change agents. In some cases, the change process happens

better as a grass-root movement run by the developers as opposed to a top-down process from the management. Having the persons that will use the process and the tools themselves on-board is important in order to establish sufficient motivation and to build necessary hands-on knowledge on a detailed level.

A Radical Change Costs Extra Resources and Needs Support from Management The development team has to tackle two challenges at once—they have to understand, adapt and implement the SafeScrum® process and they have to develop a new product. This adds extra costs, uncertainty and risk. It is important that the management support this approach and the work that is done. This gives the team freedom and time to try out new ideas as well as financial resources to establish an efficient tool chain that is tailored to support the development process. Without this support and these resources, it is difficult to succeed.

External Support and Validation Strengthens the Change If the company already uses agile development and wants to move into development of safety-critical software, it is easier to start using SafeScrum® on your own. If you are not using any agile development process already you should hire personnel with the relevant expertise or use a consultant. Although the changes may be done bottom-up (by the team), external input on methodology and safety assessment are needed to enable the team to prioritize change actions, discuss ideas and evaluate the suitability of SafeScrum® and compatibility with the IEC 61508:2010 standard, which is important to maintain. In addition, having external expertise on agile processes involved from the start will also play an important role when establishing support from management throughout the change process. It will make the pioneer project more visible and interesting to other parts of the organization.

Tools Are as Important as Processes When developing software that has to undergo detailed assessment by a certification body, there is a need to provide extensive documentation of process compliance and traceability of the process—also known as Proof of Compliance or PoC. Traditionally, this causes a lot of effort to be spent on producing documentation in addition to the software development itself. SafeScrum® works better here because tools automate or support the team in creating this type of information as a by-product of development. For example:

- Code reviews and traceability is enabled through the daily work by the developers that have to do little extra work to produce documentation. See also Chap. 8.1—"Requirements-story-code-test traceability".
- Code documentation is embedded in or coupled to the code itself, something that is of direct value to the developers when they share or review each other's code.

Change Needs to Be Done Step-by-Step First and foremost—start with simple systems. In this way, using the new process can start in the simplest possible way, using only the core elements—the standard Scrum activities. When this works, we can move on to more complex systems and problems. If we work in this way, we will be able to frequently evaluate and refine the process, potentially with external support, for example, for safety analysis. This approach also gives the team confidence as they always have a working process.

4.6.3 System Architecture

Architecture has received too little attention in agile development. The agile idea was originally that the architecture should "grow out" of the iterations but this idea has now been dropped by most projects. In some companies, they have introduced sprint zero, which is used to experiment with several architectures in order to find the best solution. We will not recommend any special approach but will offer the following advice:

- An architecture that satisfies the relevant safety standard should be ready before coding starts. There are two reasons for this: (1) it is extremely costly to change architecture once we have started to code and build the system and (2) we need to get the safety requirements early and for this we need the system's architecture and how it influences and is influenced by its environment.
- Consider at least two alternatives—for example, selected among documented architectural patterns [7], or use an analysis method such as ATAM (Architecture Trade-off Analysis Method) to make an informed choice—see [10].
- Document the choice. This is important for several reasons:
  - Writing down the decision forces you to think it through.
  - The document can be read by others who then have the opportunity to comment or disagree.
  - Select or suggest better solutions.
  - It will make the whole process transparent—also for the assessor.

We can make good architectural decisions based on the epics—what the system will do plus a small set of architectural patterns. For one large family of safety-critical systems—the control systems—the natural architectural choice is the observe–react pattern. This general pattern has been used in a wide variety of systems—from car breaks (ABS) to flight control—see an example in the diagram below (Fig. 4.10).

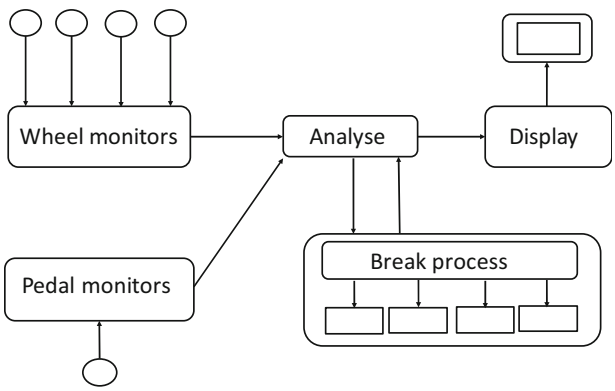

Fig. 4.10 Example of the use of the observe–react pattern

In addition to the observe–react pattern, the model-view-controller pattern and the publisher–subscriber patterns are also used in safety-critical control systems—see Chap. 4.6.4 and Annex C.

When describing an architecture, the following issues—copied from ISO 26262:2010, part 6—should be considered and described:

1. The static design aspects of the software components, which address:
   (a) The software structure including its hierarchical levels
   (b) The logical sequence of data processing
   (c) The data types and their characteristics
   (d) The external interfaces of the software components
   (e) The external interfaces of the software
   (f) The constraints including the scope of the architecture and external dependencies
2. The dynamic design aspects of the software components, which address:
   (a) The functionality and behaviour
   (b) The control flow and concurrency of processes
   (c) The data flow between the software components
   (d) The data flow at external interfaces
   (e) The temporal constraints

To determine the dynamic behaviour (e.g. of tasks, time slices and interrupts) the different operating states (e.g. power-up, shutdown, normal operation, calibration and diagnosis) should be considered. To describe the dynamic behaviour (e.g. of tasks, time slices and interrupts) the communication relationships and their allocation to the system hardware (e.g. CPU and communication channels) should be specified. In the case of model-based development, modelling the structure is an inherent part of the overall modelling activities.

It is important to be aware that all the activities included in lists (1) and (2) above have to be done upfront using all information that is currently available. However, this does not imply that they cannot be changed later if the environment or the customers' requirements change. Thus, the flexibility achieved by using SafeScrum® or any other agile development method is not that you do not need to do the job upfront but that you have a process that will enable later changes.

4.6.4 UML in Safety-Critical Software: Two Examples

According to IEC 61508:2010, the use of a formal or semi-formal design method is highly recommended (HR) for projects that develop software for SIL 3. According to this standard, UML is one of the accepted semi-formal methods. There are several tools that support UML—for example, Rhapsody. Several of the available UML tools also help both with modelling and with generating part of the code. For a complete overview of UML, see the UML bible [15].

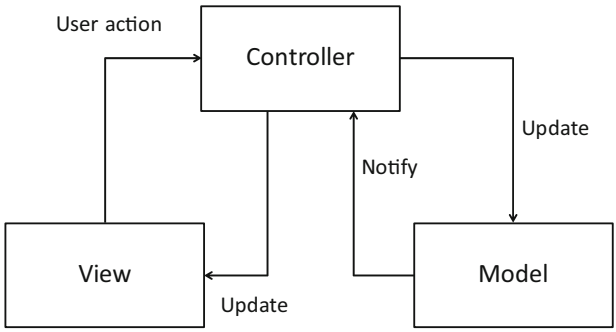

Fig. 4.11 Model-View-Controller pattern

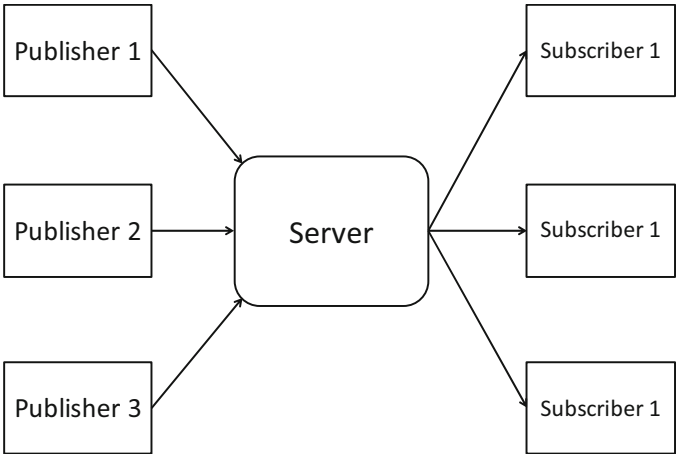

Fig. 4.12 Server–Subscriber pattern

Although UML is an extremely rich language, only a few of the diagrams are used by our industrial partners—namely state diagrams, class diagrams and sequence diagrams. The state diagrams, however, are often replaced by the Model-View-Controller (MVC) pattern—see Fig. 4.11—or sometimes with the server–subscriber pattern—see Fig. 4.12.

One of our industrial partners uses UML diagrams throughout the development process, starting with informal sketches on paper or a whiteboard during the specification and design phases. It is important to keep the diagrams simple as far as possible. The diagrams are later elaborated into complete and syntactically correct diagrams during development, now by using a tool. The possibility to use this iterative process is considered one of the important features of UML.

Class diagrams and sequence diagrams are used together—class diagrams and later, object diagrams, to show how things are connected and sequence diagrams to show how it works. All diagrams that are developed should be updated throughout

the development process and made available for the assessor of the final review. Although there are several reports on the efficiency of performing early FMEA based on UML diagrams—for example, the sequence diagram—this is seldom done.

UML and Safety Analysis
Sequence diagrams are useful when we do safety analysis, both informal and formal. The reason is easy to see—the sequence diagram shows the system in action—message passing, action alternatives, active objects and so on. The timeline of each object shows the object's inputs and outputs. Thus, it is easy to analyse the behaviour of each object by asking some simple questions, most conveniently documented in an FMEA form (see Annex B.6)—one for each object. A typical set of questions to ask—possibly interpreted as object specific failure modes—could be: What happens if

- Input A is not handled or contains wrong or incomplete info?
- Output B is sent too late, not sent or contains wrong or incomplete info?

The answers to the questions above will help us identify barriers in the system and thus make it safer.

UML and Agile Development
Whether you can combine agility with UML or not is hotly debated among developers in fora such as blogs. Daniels [6] presents the opposing side in the argument with the following challenge where he contrasts the agile manifesto with some perceived consequences of using UML:

- *"Individuals and interactions over processes and tools.*
  UML and its supporting tools are the cornerstone of my detailed and rigorous development process.
- *Working software over comprehensive documentation.*
  With UML I can spend years documenting my software.
- *Customer collaboration over contract.*
  UML lets me freeze the requirements early.
- *Responding to change over following a plan.*
  Argh! I'll have to redraw all those nice UML diagrams".

After this, however, he puts it all in perspective when he remarks that *"UML is just a language"*. As such, it is neither good nor bad for an agile process—it all depends on how and for what purpose you use it.

Shannon [16] argues against using UML in agile development. Her arguments are mostly related to the tool side and run as follows: *"We could just take a look at how teams work and the problems they face while using desktop modelling tools such as:*

- *The tools are fairly complex to use, take a long time to install and setup*
- *Sharing models is complicated.*
- *Working together on the same model remains impractical.*
- *Generating code is tedious and sometimes useless.*
- *Questions remain around synchronizing the 'code/model'.*
- *A simple tool dedicated to hosting and managing versions of models doesn't exist".*

Here focus is on the UML tools. If you drop tools throughout development and just use them to produce the final documentation, Shannon's arguments are not so important anymore. If we instead start with simple sketches for discussion, UML can help us to

- Generate ideas for design solutions at all stages of the development process
- Communicate with the customer and with other developers

Remember that UML is a model of a part of the application domain, not a model of the software alone. Keeping the UML model on paper or on a whiteboard makes it easy to share and to cooperate on the modelling.

When we agree on the model, we can use a tool to store it in digital form to be used, for example, for documentation. As any other documentation, it needs to be updated when something is changed. This might be tedious if we use a tool to generate code based on the model. On the other hand, we can be sure that the model and the code are synchronized.

4.6.5 Coding Standards and Quality Metrics

One of the challenges when it comes to coding in a safety-critical project is the coding standard. The project must have a coding standard both according to the safety standard and in order to improve communication within the team. In addition, a coding standard will make it easier to perform code reviews and to maintain code written by others.

A programming language coding-standard shall:

- Specify good programming practice.
- Proscribe unsafe language features—constructions that should not be allowed or only allowed under specific, documented circumstances.
- Promote code understandability. This is important for code reviews, for example.
- Facilitate verification and testing.
- Specify procedures for source code documentation.

Where practicable, the following information shall be contained in the source code:

- Legal entity—for example, company and authors.
- Description—what does this chunk of code do and how does it do it?
- Inputs and outputs—names, types and their meaning.

Some standards—for example, IEC 61508:2010—want to eliminate or reduce the use of pointers, recursive code and such like. This does not mean that pointers are forbidden. What it means is that you should document where they are used and the reason they are needed.

It is important to control code complexity. It is also a requirement in several standards. The method needed to do this can vary from an advanced metrics regime to a simple process where somebody assesses the code as OK or being too complex based on his or her experience. Some of the metrics used in industry are Henry-Kafura's fan-in fan-out metrics [9] and McCabe's cyclomatic value—v (G) [12]. Note that there has been a lot of criticism levelled at McCabe's cyclomatic number. Even so, it is used a lot in industry—not for prediction of error density or content but as an indication for code complexity.

Besides the problem of choosing one or more metrics, we are also faced with the problem of choosing an action limit. We do not achieve complexity control by using McCabe's cyclomatic number if we at the same time do not define a limit for this number. We might, for instance, use a rule such as "If v(G) is greater than five, the developer shall either rewrite the code to reduce the value or write a short note explaining why the higher-than-normal v(G) is permissible here". We may use the rules defined by others or use these rules as a starting point and modify them as we gain experience.

Another important metric is the module size, which is important for two reasons: it sets a limit to the number of code lines a developer has simultaneously "to keep in his head" and it will decide the lowest level for traceability. As an example, we will consider IEC 61508-7:2010, appendix C 2.9. We asked a representative from a European certification organization to give us a recommended size for subprograms and modules and got the following response:

- *Subprogram sizes should be restricted to some specified value, typically, two to four screen sizes.* This gives a subprogram size of 200–400 lines of code.
- *A software module should have a single well-defined task or function to fulfil.* This definition allows for several interpretations. We recommend the size not to exceed 1000 LOC for modules in order to have a clearly arranged and structured software architecture.

The same European certification organization, however, does add an important remark: *"In general we interpret a module as a set of code which fulfils a defined function; this makes also sense from a testing point of view (test specification level)"*. Furthermore, *"...for us it is more important to have a well-structured architecture with defined function modules than to insist on defined LOC restrictions"*.

See Chap. 7.7 for further details.

4.6.6 Configuration Management (CM)

It is always a challenge to change software—safety critical or not. Change is always tightly connected to configuration management. The challenge is more important for agile development than for any other development paradigm since agile development promises to "embrace change". In addition, the need for configuration management will increase when we use agile development since changes will be more

frequent—possibly several changes in each sprint. Changes during agile development come from several sources, for example:

- New requirements added after the development process has started
- Changes to existing requirements due to new knowledge or new customer needs
- New risk and hazards due to changes in the operating environment or new knowledge
- Refactoring—tidy up the code, which is important in agile development
- Not-accepted user story implementation from a sprint

All changes, irrespective of source, represent challenges for both the developers and for the system's integrity, for example:

- Testing. Which tests need to be re-run after the changes—the need for regression testing has to be evaluated
- Change impact analysis—see Chap. 8.2: How will the change affect system?
  - Complexity—both IEC 61508:2010 and EN 50128:2011 require that the system complexity shall be controlled.
  - Safety—which safety and hazard analyses should be checked or repeated?

When the purpose of the change is to correct an error, it is also important to check that we have attacked the root cause of the error and not just its symptoms and not introduced new errors or re-introduced old ones.

The CM (Configuration Management) process is well known and there is a plethora of tools available to support it. Traditionally, the processes have been heavy on management and documentation. None of these concepts fit well with agile development.

An important statement related to CM is that the Software Quality Assurance Plan, Software Verification Plan, Software Validation Plan and Software Configuration Management Plan shall be drawn up at the start of the project (i.e. before the first sprint) and be maintained throughout the software development life cycle. The important thing for SafeScrum® is to have a procedure at the start of each sprint where all plans are updated if necessary. This can be done either by the SafeScrum® team itself as part of the sprint planning process or by the people who developed the plans, using information from the SafeScrum® team—the alongside engineering team.

Some standards require that all information related to testing—for example, environment, tools and software—shall be included in CM. However, testing done during development using, for example, test-first development does not need to be included. The reason for this is that these tests are written by the developer on the fly to check the current code chunk. The tests are usually changed several times during development and a rigid CM apparatus may slow down the process without contributing anything to the quality of the software. However, if you are planning to use unit tests or tests developed for TDD later to include them into the system tests, they need to be included in the CM system.

In most projects, documented testing only includes integration testing and system testing. The most important goals of CM are to:

- Have administrative and technical control throughout the life cycle.
- Apply the correct change control procedures and document all relevant information for safety audits—that is, that the CM job is done properly.
- Have control over all identified configuration items.
- Formally document the releases of safety-related software.

An important challenge to the SafeScrum® process is the first statement: administrative control throughout the lifecycle. For the other CM requirements, the challenge for SafeScrum® is not to fulfil the requirements but to decide how often and under what circumstances they should be fulfilled. Most of the information needed for efficient CM is created automatically by tools. We suggest the following approach:

- Management decides at which milestones a new configuration should be defined. This is done before the project starts and is described in the CM plan.
- The responsibility for managing the CM is normally assigned to the quality assurance department (QA).
- All code and data are tagged during check-in. The tags are administrated by the QA but used by the SafeScrum® team.

When developing safety-critical systems, changes may have effects that are outside the changed modules or components. This challenge is handled by change impact analysis. Even though this is important, it is not part of CM.

4.6.7 Synchronizing SafeScrum® and a Stage-Gate Process

A stage-gate process is a development process constructed using a set of clearly defined and separated activities called stages. There is a control gate between each connected pair of stages. These control gates are used to check that the results from one stage fulfil the requirements of this stage and that it is usable for the next one. The Cooper stage-gate model [5] operates with the following stages: Discovery (of a need), Scoping, Business case, Development, Testing and validation, Launch and Post launch review. In some ways, the stage-gate model is more like a business decision model than a software development model. Both scoping and building business cases are activities outside software developers' knowledge and interest.

If we stick with the idea of SafeScrum® and the concept of separation of concerns, we would say that only "Development and Testing" and Validation are inside the Scrum domain and that the rest could be left as it is. This would, however, not make sense since changes to the software, for example, could require a new business case.

One of the most thorough works on stage gates and agile development is done by Sommer et al. [17]. The following discussions are based on their work. Their first observation, which is relevant for all introductions of any new processes or methods,

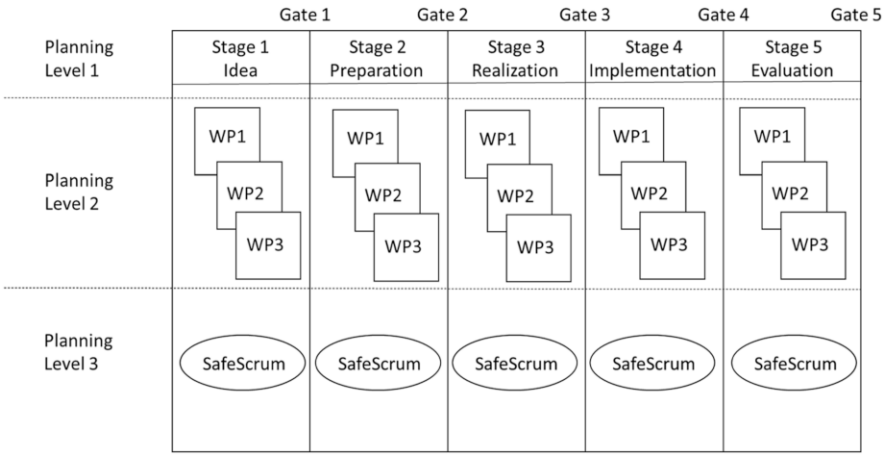

Fig. 4.13 The PRINCE2 model

is that methods that do not change the company culture will have little or no effect on important parameters such as quality and cost-effectiveness. This also holds for introducing stage-gates. A practical model for combining stage-gates and agile development is a project management standard developed by the British government called PRINCE2 [2]. This model fits well with the approach suggested by Sommer et al. [17]. The following diagram describes the model adapted to SafeScrum® (Fig. 4.13).

In this model, the work packages (WPs) are template-based documents stating the deliverables from each employee and team. The templates are developed at the start of each stage. Each stage will contain a series of Scrum sprints.

It is customary to use a three-level approach. Activities at these three levels should be aligned. The Scrum activities at each stage will contain several sprints. This approach has been enhanced by Sommer et al. [17] with extra information as follows:

- Strategic level (level 1)—strategic planning and decision-making, which contains the planning level for the product portfolio management and steering committee.
- Tactical level (level 2)—weekly resource planning, and tactical planning between product development teams and the operational organization. Focus is on the value chain and the project portfolio coordination. The stakeholders from across the organization meet physically to coordinate resources. Stakeholders included here are project management, sales and market, production and quality assurance.
- Execution level (level 3)—day-to-day decisions.

Sommer et al. [17] also suggest that a Scrum team is used for the feasibility study. This study contains the following activities and ends with a go/no go decision:

- Refine product vision.
- Develop product backlog and prototype. In agile development, this is often called sprint 0.

- Design workshop.
- Risk workshop—identify project risks.
- Budget workshop.

This process will need several sprints. The number of sprints needed will depend on the complexity of the study.

References

1. *The Business Dictionary*. 2017 [cited 2017]; Available from: http://www.businessdictionary.com/
2. Bentley, C. (2010). *Prince2: A practical handbook*. New York: Routledge.
3. Bjerke-Gulstuen, K., Larsen, E.W., Stålhane, T., & Dingsøyr, T. (2015) High level test driven development – Shift left. In C. Lassenius, T. Dingsøyr, & M. Paasivaara (Eds.), Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25–29, 2015, Proceedings. Cham: Springer International Publishing, pp. 239–247, 978-3-319-18612-2.
4. Bukowski, J. V. & Chastain-Knight, D. (2016) Assessing safety culture via the site safety index (TM). In *Proceedings of AIChE 12th Global Congress on Process Safety*. Houston, TX: Exida.
5. Cooper, R. G. (2011). *Winning at new products: Creating value through innovation*. New York: Basic Books.
6. Daniels, J. (2003). *Modelling in an agile world*. Fastnloose ltd, p. 15, http://docplayer.net
7. Friedrichsen, U. (2016). *Resilience reloaded – More resilience patterns*. slideshare.com
8. Grote, G., & Künzler, C. (2000). Diagnosis of safety culture in safety management audits. *Safety Science, 34*(1), 131–150.
9. Henry, S., & Kafura, D. (1981). Software structure metrics based on information flow. *IEEE transactions on Software Engineering, 5*, 510–518.
10. Kazman, R., Klein, M., & Clements, P. (2001). *Evaluating software architectures-methods and case studies*. Boston: Addison-Wesley.
11. Laukkarinen, T., Kuusinen, K., & Mikkonen, T. (2018). Regulated software meets DevOps. *Information and Software Technology, 97*, 176–178.
12. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 4*, 308–320.
13. Moe, N. B., Aurum, A., & Dybå, T. (2012). Challenges of shared decision-making: A multiple case study of agile software development. *Information and Software Technology, 54*(8), 853–865.
14. Needle, D. (2010). *Business in context: An introduction to business and its environment*. Andover: Cengage Learning EMEA.
15. Pender, T. (2003). *UML bible*. New York: Wiley.
16. Shannon. (2014). *4 Reasons Why UML is Agile (or Could Be)*, in *GenMyModel*.
17. Sommer, A. F., Hedegaard, C., Dukovska-Popovska, I., & Steger-Jensen, K. (2015). Improved product development performance through agile/stage-gate hybrids: The next-generation stage-gate process? *Research Technology Management, 58*(1), 34–44.

Chapter 5
Standards and Certification

What This Chapter Is About

- *We give a short introduction to the role standards play in development of safety-critical software and look at what standards are not.*
- *We discuss the challenges the developers meet from the standards.*
- *We discuss shortly the software certification process.*
- *We give a short overview of the responsibilities of the development organizations, the developers and the assessors.*
- *Information to be delivered to the assessor and when the information shall be delivered.*

5.1 The Role and Importance of Standards

This chapter is only an introduction to the safety standards in general and to the IEC 61508:2010 series in particular. It will guide you to the important parts and show you the most important issues to read and to use.

Standards, as they are used in the development of safety-critical systems, are used to make sure that a defined minimum set of activities are employed during design, development, analysis, testing operation and maintenance. Several standards—for example, IEC 61508:2010—are process-oriented. That is, they do not just say what shall be done but also in which sequence it shall be done. Due to a more or less goal-based approach, most safety standards say that a certain activity shall be performed but not in which way, how much or how often. This is a challenge when developing safety-critical software. IEC 61508-5:2010 and -6 includes, however, some guidance. The IEC 61508:2010 standard has seven parts and is organized as follows:

- IEC 61508-1: General requirements
- IEC 61508-2: Requirements for electrical/electronic/programmable electronic safety-related systems

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_5

- IEC 61508-3: Software requirements
- IEC 61508-4: Definitions and abbreviations
- IEC 61508-5: Examples of methods for the determination of safety integrity levels
- IEC 61508-6: Guidelines on the application of IEC 61508-2 and IEC 61508-3
- IEC 61508-7: Overview of techniques and measures

Since a standard is only updated every 5–10 years, the processes, methods and techniques specified in the standards may be outdated over time. Some methods are replaced by more efficient ways of doing things. Others are just plain unnecessary since they are the commonly used approach anyhow, or are performed by automatic tools, used almost everywhere. A good example is the standard's requirement to use structured programming and a modular approach—nobody does it any other way these days.

Despite their shortcomings, standards are an import tool in business, for example, to improve communication between customers and developers; when something is defined in a standard, we have a common ground for communication.

5.2 What the Standards are Not About

Even though the standards regulate many important activities—see below—there are several important areas, which are not covered by the safety standards—at least not directly. The most important ones are:

- **Project management.** This is neither described nor required by the standards. On the other hand, without good project management, no standard will help you deliver a good product.
- **Project organizations.** Since some of the safety standards define a certain distribution of roles, the standards to a certain extent will influence your project organization. See diagram below from EN 50128:2011 (railway). The big box, containing, for example, validator and verifier roles, contains company-internal roles. IEC 61508:2010, on the other hand, does not discuss roles at all (Fig. 5.1).

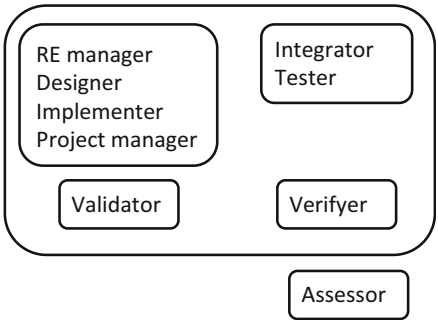

Fig. 5.1 An example of project roles according to EN 50128:2011

- **Communication.** Even though communication, both inside the project and between the project and the customers, is one of the most important aspects of software development, the standards do not touch this topic at all. Each project or organization needs to find its own rules and ways of communication. However, standards will, as mentioned earlier, improve and enable communication.
- **Certification.** We do not always have to certify a system as this is not required within all domains, even if it is safety-critical. Anyway, the IEC 61508:2010 standard does not require certification— it is up to the customer or the authorities. In some cases, the customer or an authority (e.g. within the railway domain) will require certification.

5.3 The Process of Product Certification

The product certification can be done in several ways—not all of them equally confidence building. The two extremes are: (1) check that all required activities have been done—the checklist approach (without focusing on the complete system)—and (2) go through all requirements and recommendations to check that they are adhered to in an appropriate way, that the intended use is taken care of, for example, that sufficient resources are used. In addition, the assessor may check personnel qualifications. The assessor's rationale for using the most severe alternative—alternative 2—is usually that they put their reputation at stake when they certify a system and they are afraid to certify something that is not up to standards.

5.4 On Standards for Safety-Critical Software

The first activity needed when developing safety-critical software is to decide the level of criticality—that is, the risk incurred when the system is put into operation. This is called SIL—short for Safety Integrity Level. Although there are several standards for how to assess the risks related to a system's operation, many customers require a certain risk level, irrespective of what a proper risk analysis might end up with. There are two important reasons for this:

- The customer wants a high SIL as part of his sales drive—"Our systems are built to the most stringent safety requirements".
- An external authority—for example, a government department—has set a safety integrity level based on political or economic considerations.

In IEC 61508:2010, criticality is defined as a SIL, which is a number from 1 to 4, even though some organizations also operate with a SIL 0 or level a, meaning "no special actions are needed". Figure 5.2 describes how the safety integrity level is

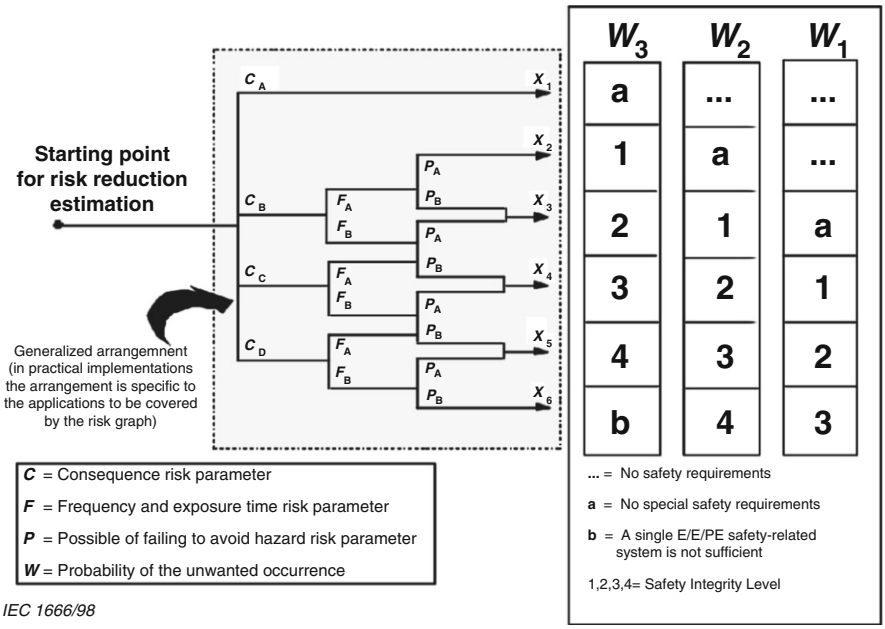

Fig. 5.2 SIL assignment according to IEC 61508-5:2010 (E. 2)

decided in IEC 61508-5:2010, figure E.2. The process is simple: assign values to the three parameters:

- Consequence risk parameter (C)
- Frequency and exposure time risk parameter (F)
- Possibility of failing to avoid hazard risk parameter (P)

This will give you an X-value. Next, choose one out of three values for the probability of the unwanted occurrence—W. The X-value and the W-value will then indicate the required SIL-value (Fig. 5.2).

From this diagram, we see that the following factors influence the safety level: the consequences, frequency and probability of an unwanted incident plus the probability that the consequences cannot be avoided. Other standards may use other parameters, but the ones used here are quite common. We see from the diagram that if failures, for example, have only small consequences, then special safety requirements are not needed.

The SIL requirements are only mandatory for the safety functions—that is, the functions that are needed to take care of the system's safety concerns. The IEC 61508-4:2010 defines a safety function as follows:

> *"**Safety function:** function to be implemented by an E/E/PE safety-related system or other risk reduction measures, that is intended to achieve or maintain a safe state for the EUC (Equipment Under Control), in respect of a specific hazardous event"*

5.5 Development Challenges Related to Safety Standards

The process and methods required for the development of safety-critical software will, to some extent, be specified by the standard. The level of details in these requirements will vary. The high-level requirements for software in IEC 61508:2010 are summed up in part 3, clause 7 as follows:

- *"A safety lifecycle for the development of software shall be selected and specified during safety planning in accordance with Clause 6 of IEC 61508-1 (Management of functional safety).*
- *Any software lifecycle model may be used provided all the objectives and requirements of this clause (clause 7) are met.*
- *Each phase of the software safety lifecycle shall be divided into elementary activities with the scope, inputs and outputs specified for each phase*
- *Provided that the software safety lifecycle satisfies the requirements of Table 1, it is acceptable to tailor the V-model (see Figure 6) to take account of the safety integrity and the complexity of the project.*
- *Any customisation of the software safety lifecycle shall be justified on the basis of functional safety.*
- *Quality and safety assurance procedures shall be integrated into safety lifecycle activities".*

Two issues are important here: (1) any life cycle model may be used as long as it satisfies the standard's requirements and (2) the choices made must be justified. We recommend that this is done in agreement with the assessor before commencing on development.

The safety standard's detailed requirements for the development process are organized in tables in Annexes A (normative) and B (informative) of IEC 61508-3:2010—one table for each part of the development process, parameterized by the SIL value. The requirements belong to one out of four classes—"---" (no recommendation neither for nor against), NR (not recommended), R (recommended) and HR (highly recommended). Only the requirements marked with HR are compulsory—or at least almost compulsory. It is possible to argue in a goal-based manner—that is, to explain that the alternative technique or measure will achieve the same goal. The argument can be as follows: "The purpose of this requirement is to achieve A and B. Instead of following the stated requirement, we will do something else which will allow us to achieve the same goals". It is, however, up to the certifying organization to accept or reject this.

When deciding which techniques and measures to apply, a practical approach is to look at the aim for each technique and measure in IEC 61508-7:2010 (see Fig. 5.3). This makes it also easier to develop an argument if another technique or measure is used.

Table 5.1 shows an example from IEC 61508-3:2010 Annex A, which shows what is needed for a software safety requirements specification, parameterized by the SIL value (Table 5.1).

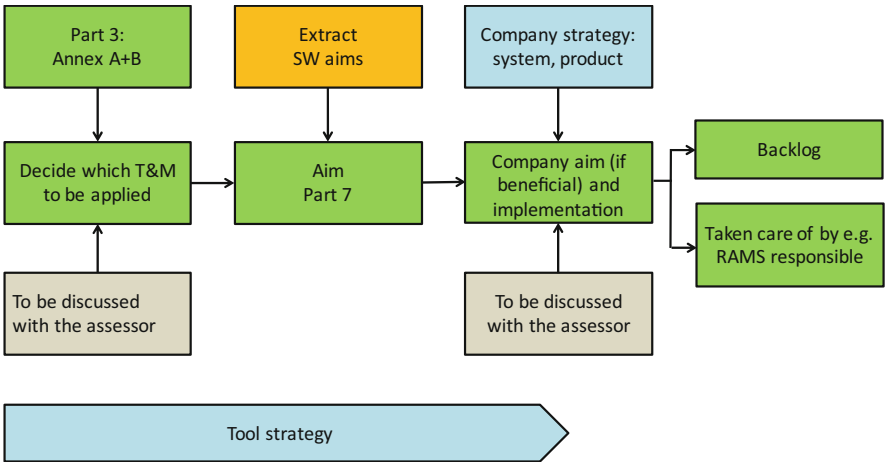

Fig. 5.3 Techniques and measures and the related aim

Table 5.1 IEC 61508-3:2010 Table A1—Software safety requirements specification

Technique/Measure[a]		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1a	Semi-formal methods	Table B.7	R	R	HR	HR
1b	Formal methods	B.2.2, C.2.4	–	R	R	HR
2	Forward traceability between the system safety requirements and the software safety requirements	C.2.11	R	R	HR	HR
3	Backward traceability between the safety requirements and the perceived safety needs	C.2.11	R	R	HR	HR
4	Computer-aided specification tools to support appropriate techniques/measures above	B.2.4	R	R	HR	HR

NOTE 1 The software safety requirements specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application
NOTE 2 The table reflects additional requirements for specifying the software safety requirements clearly and precisely
NOTE 3 See Table C.1
NOTE 4 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7
[a]Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following that number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application

According to this table, there are only recommendations for requirements to the software safety requirements specification if you have assessed the necessary safety to be SIL 1 or SIL 2. As soon as you move up to SIL 3, semi-formal methods,

Table 5.2 IEC 61508-3:2010 Table B.1—Design and coding standards (referenced by Table A.4)

Technique/Measure[a]		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Use of coding standard to reduce likelihood of errors	C.2.6.2	HR	HR	HR	HR
2	No dynamic objects	C.2.6.3	R	HR	HR	HR
3a	No dynamic variables	C.2.6.3	---	R	HR	HR
3b	Online checking of the installation of dynamic variables	C.2.6.4	---	R	HR	HR
4	Limited use of interrupts	C.2.6.5	R	R	HR	HR
5	Limited use of pointers	C.2.6.6	---	R	HR	HR
6	Limited use of recursion	C.2.6.7	---	R	HR	HR
7	No unstructured control flow in programs in higher-level languages	C.2.6.2	R	HR	HR	HR
8	No automatic type conversion	C.2.6.2	R	HR	HR	HR

NOTE 1 Measures 2, 3a and 5. The use of dynamic objects (e.g. on the execution stack or on a heap) may impose requirements on both available memory and also execution time. Measures 2, 3a and 5 do not need to be applied if a compiler is used, which ensures a) that sufficient memory for all dynamic variables and objects will be allocated before runtime, or which guarantees that in case of memory allocation error, a safe state is achieved; b) that response times meet the requirements
NOTE 2 See Table C.11
NOTE 3 The references (which are informative, not normative) "B.x.x.x", "C.x.x.x" in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7
[a]Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following that number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application

forward and backward traceability between safety requirements and safety needs, plus a computer-aided specification tool are all highly recommended.

A real-life example runs as follows: Table B.1 (Table 5.2) has strict requirements on the use of dynamic variables, interrupts and pointers for SIL 3 and SIL 4. The requirement "Limited use" is commonly interpreted as a requirement to document all uses of pointers, for example, in a table and explain why we need to use a pointer in each case. One company did, however, consider this to be too much work—especially since such a table would have to be maintained throughout the development process. They thus asked the assessor organization if they could use the following approach instead:

"We try to avoid dynamic allocation in safety-critical components. This is achieved by using a predefined set of design patterns, described in our coding standard.
Where we need to use dynamic allocations, we want to catch bad allocations by the use of exceptions. As a rule, these exceptions will cause a restart of the affected component. If this happens several times in a row, it triggers a system restart.
Pointers are used according to a set of coding patterns—for example, pointer initialization, preventing pointer arithmetic, and checking of the pointers using an "assert" plus check for zero-pointers."

This was accepted without further ado by the assessor's organization. Note that even though Annex B in IEC 61508-3:2010 is just informative, several assessor organizations—for example, TÜV certification bodies—consider it to be normative. On the other hand, it might also be argued that all the process requirements in IEC 61508:2010, and in most other standards for the development of safety-critical systems, are just sound software engineering practices. There are also research published that claim that if we develop all software to SIL 3 or SIL 4 it will cost more initially—an investment—but we will get the return of investment later, during maintenance [1]. One of our industrial partners has stated, *"Any software that is expected to live more than five years should at least be developed according to SIL 3."*

5.6 The Developers' Responsibility

It is the developer's responsibility to make sure that the software is developed according to the standard's requirements. Since the assessor is not present all the time to check that the development process defined by the level of safety criticality is followed, he needs PoC—Proof of Compliance/Conformance. It is the developers' responsibility to create PoC or the information needed to create PoC. This does not necessarily mean that the developer shall write down the information somewhere. On the contrary, it means that the developers as often as possible should use tools that leave traces such as logs. Several standards, for example, the EN 5012x series for railway and ISO 26262:2010 series for automotive standards, require a safety case to be developed. An effective approach is to develop an agile safety case [3] (Fig. 5.4).

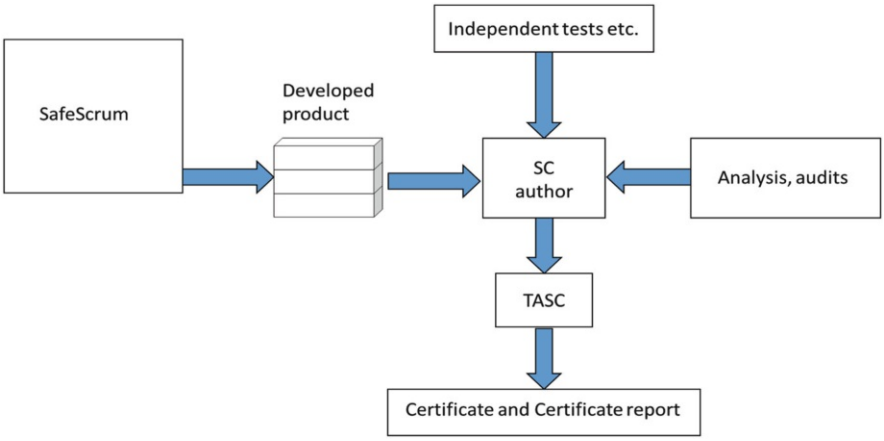

Fig. 5.4 The relationship between the safety case and SafeScrum®

5.7 The Assessor's Responsibility

The assessor's responsibility is to check that the development process is according to the required standard. There is, however, a large difference between what the assessors consider important and how they check it. There is also a variation in what is checked and how it is checked—some only check compliance with the requirements in the normative parts of the standard, while others also require compliance with the informative parts. If the developer delivers a well-documented safety case, this will be the most important document for the assessors.

Since the IEC 61508:2010 series was issued for the first time in 2000, a certification system for safety-related EEPES (electrical/electronic/programmable electronic systems) has been put into practice by a few certification bodies without any harmonization [2].

Last but not least—the assessor is not responsible for any problems related to safety or anything else. The company that brings a product on the market carries the full responsibility of the product's behaviour. The certificate is not a warranty for the product but a warranty that the prescribed processes have been followed by the producing company.

5.8 The Development Organization's Responsibility

The development organization's first responsibility is to keep its promises to the customer. For safety-critical software, this includes both functions and the process requirements specified by the standard. The first part is included in the customer's SAT (Site Acceptance Test), while the latter is done first by the QA responsible and later assessed by the assessors.

In order to make the assessment process work, it is important to leave information that can be used as PoC for each required process step. This will usually not have to be a formal report. Such things as screen-shots, printout and logs will suffice in most cases. Any information that can be used to convince the assessor that a particular method has been used or that a particular process step has been performed can be a PoC. It is not the formality or document layout that makes it a PoC but the information it conveys to the assessor.

A set of documents, for example, specifications and records, is frequently called "documentation". Some requirements for documents—such as the requirement to be readable—relate to all types of documents.

It is important to bear in mind that according to the definition shown in Chap. 4.5—"Information items", for example, a set of snapshots using a discussion on a whiteboard, can be used as documentation for system architecture or design. In addition, the document needs a unique ID. At least one assessor has added the requirement that in order to be considered documentation, the document—whatever it is—should contain a date and the names of all who participated in making the

document. However, there can be different requirements for specifications (e.g. the requirement to be revision controlled) and for records (e.g. the requirement to be retrievable).

Cooperation between assessor, Scrum master, developers, the RAMS engineer and the product owner is an essential part of SafeScrum®. This cooperation is used to assure that what we do will later be accepted by the assessor. We should not ask an assessor how to do something but instead tell him what we will do and ask if this is acceptable. Such cooperation will reduce the assessor's work during certification and is thus a win–win situation.

References

1. McDermid, J., & Kelly, T. (2006). Software in safety critical systems-achievement & prediction. *Nuclear Future, 2*(3), 140.
2. Myklebust, T. (2013). Certification of safety products in compliance with directives using the CoVeR and the CER methods. In *proceedings of ISSC*. Boston: Springer.
3. Myklebust, T., & Stålhane, T. (2018). *The agile safety case*. Berlin: Springer.

Chapter 6
The SafeScrum® Process

This chapter lays out the basis for SafeScrum® and discuss the iterative and incremental development process. In addition, we describe the details in SafeScrum®, such as:

- *The associated roles.*
- *Fundamental SafeScrum® concepts.*
- *How to prepare a SafeScrum® project.*

6.1 SafeScrum® in Perspective

SafeScrum® is a process framework describing how software engineering of safety-critical software may be organized and carried out in order to comply with IEC 61508-3:2010, and the standard's defined safety life cycle. However, in order to manage analysis and documentation of additional *safety aspects* as required by the standard, there are other safety-oriented activities that have to be done alongside the software development process. Hence, we will describe both *the SafeScrum® process* for software engineering, and the additional *alongside engineering process*, which is the term we have defined and that will be used throughout the book to refer to processes that are not part of SafeScrum® but that support it. When necessary, we will also describe relevant parts of Scrum, just to make the relationships between SafeScrum® and Scrum clearer.

Since this is all about safety-critical systems, we have included short descriptions of activities that are not specific to SafeScrum® or Scrum. This is necessary to get the complete picture of the needed processes and activities and includes activities such as developing a safety case and the hazard log.

Figure 6.1 presents a simplified overview of the IEC 61508-1:2010 safety life cycle and explains how SafeScrum® and alongside engineering are processes related to each other, and within the safety life cycle. See also Chap. 4.6—Preparing for

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_6

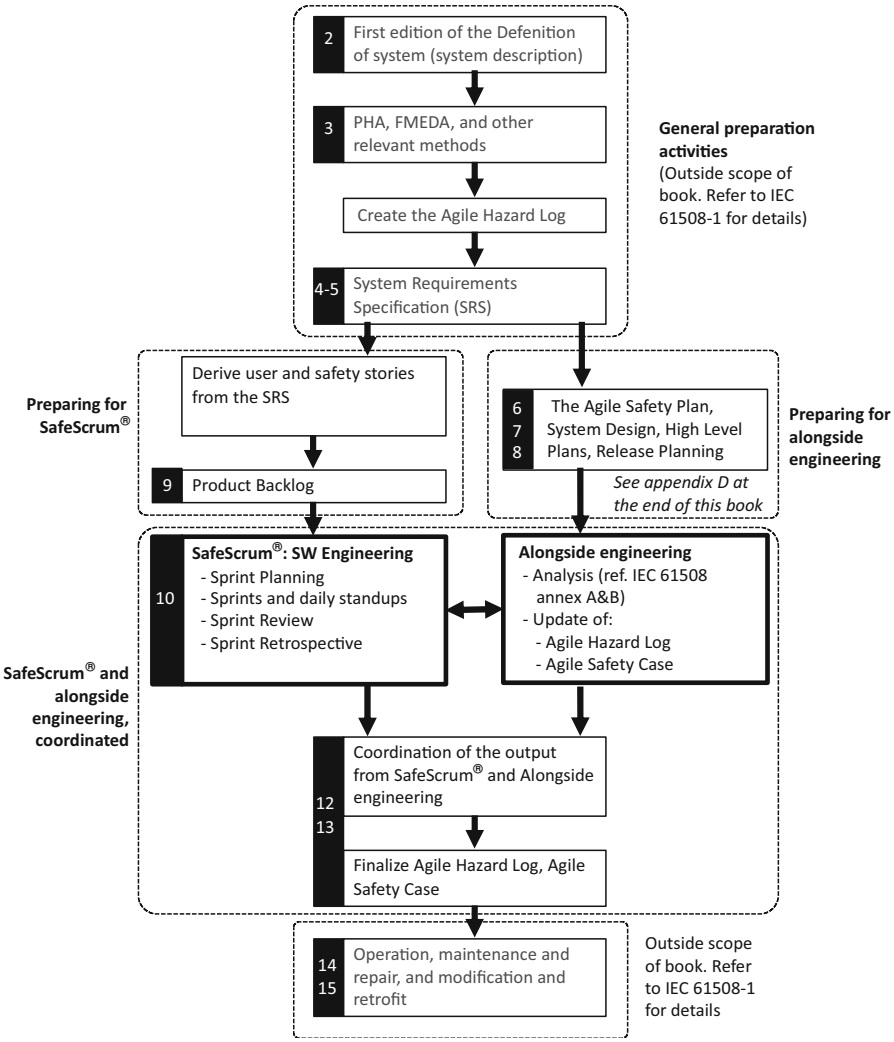

Fig. 6.1 A simplified safety life cycle model

SafeScrum®. For the agile hazard log and the agile safety case, see Chaps. 8.4.2 and 8.4.3. The coordination of SafeScrum® development and the alongside engineering team is described in Chap. 6.3.

This chapter explains the key elements of SafeScrum®: the roles, preparing for SafeScrum®, the standard SafeScrum® meetings, how to manage the sprint workflow, and how to manage requirements and traceability, etc. The process is based on plain Scrum as described by several sources, but we have added elements or made changes in order to make it support specific challenges in the development

of safety-critical software as required by the IEC 61508:2010 standard, and not at least in order to achieve certification in the end. SafeScrum® is to be considered as a framework that should be fitted to each case and context of use. Chapter 8.4 provides details on alongside engineering to support the SafeScrum® process.

6.2 An Iterative and Incremental Process

SafeScrum® inherits fundamental process features from Scrum (see Chap. 2) and is an iterative and incremental process.

An Iterative Process Work is done in iterations called sprints, which are short work periods of 2–4 weeks—see Chap. 7.2 for details. Each iteration is planned in a sprint planning meeting (Chap. 7.1) and is evaluated in a sprint review meeting (7.3). Optionally, a sprint may also be evaluated in a sprint retrospective in order to identify changes to improve the process based on recent experience (Chap. 7.4). The main motivation for working in repeated, short periods is to create a mechanism for learning and improvement, both the system under development and the process itself. Sprints can be seen as "mini projects" with room for learning and re-planning in between.

An Incremental Process The result of one or more sprints adds to the incremental growth of the final result. The functional validation is done as part of the sprint, while safety validation is done by the RAMS engineer after the sprint, as part of the alongside engineering. The results are only added to the product if they are approved. That is, that an increment meets the functional requirements *and* that it satisfies the safety requirements *and* that necessary documentation is done. Any result that is not approved (see Chaps. 7.2 and 7.3) is returned to the product backlog (see Chap. 6.5.2) to be resolved in later sprints. The basic idea is to grow the product stepwise and to ensure that the outcome from each sprint holds sufficient quality to go into the final solution. Results from each sprint should not be considered as mock-ups or prototypes, but finished code. This gives better control of what is actually finished and what is remaining in development.

6.3 SafeScrum® and Associated Roles

The roles in SafeScrum® are based on typical Scrum roles as they often are applied in development of non-safety-critical systems, but new roles are added to deal with specific requirements of the standard to address specific quality-, traceability- and safety issues. We have also taken into consideration requirements regarding independence of roles as defined in the standard or that may be required by the independent assessor. The following overview and description of roles is not absolute—it is meant as a basis for defining the roles needed in a specific project.

Some roles require specific competency and responsibility, like for example the RAMS engineer, which is a safety expert. Other roles may, in combination, be covered by the same person, for example, the person taking the Scrum master role may also serve as the quality assurer if he/she has the capacity and the load of the task is small enough. If the load of the quality assessor role is too large (e.g. in a large and complex project), it may be natural to dedicate this role and responsibility to a separate person. Also, our experience has shown that it may be valuable for the Scrum master to get involved in development or test activities to stay updated on technical details and stay close to the rest of the team. However, this depends on the specifics of the project and the knowledge, competency and capacity of the persons filling the roles.

For clarity and reference, we describe the standard Scrum roles that are part of SafeScrum®:

- **The Scrum master**: The Scrum master role should be filled by an experienced developer with insights into the technology used, the application domain and safety. The main responsibility is to facilitate the Scrum process more than leading or directing it and to ensure that safety is the number one priority throughout the process. This includes the responsibility of facilitating regular events such as sprint planning meetings and sprint review meetings, etc., and to ensure that all team members are given tasks and that problems hindering the process are solved. In order to facilitate the processes when developing a safety-critical system it is important that the Scrum master have (1) deep insight into the system under development and its requirements and (2) that he or she has a good understanding of safety engineering in general and the safety standard. This extends the Scrum master role as just facilitator, but we have seen that having a good safety understanding makes it easier to take on the role and that it creates a natural authority within the team. Thus, it will be beneficial if this role is combined with other technical roles, either developer or tester (if possible). The Scrum master will have reduced time to do, for example, development, but it will be a valuable source to detailed information and to build the relationship with the team.
- **The product owner**: The product owner is a part-time role filled by someone with an understanding of the market or customer's needs. He or she will focus on system functionality. Safety is the responsibility of the developers and the safety analysts. The product owner's main responsibility is to represent the customer or the users, either directly or as an internal proxy who interacts with the market function, etc., in the company. The product owner provides requirements and feedback on results needed to approve results and re-plan development. The product owner needs to be able to make decisions and set priorities, either directly or through consulting others, to help the team in the detailed planning of the development. As the product owner (in SafeScrum®) has a dedicated software responsibility, it is natural to collaborate and coordinate with other roles like the overall project manager, the hardware responsible, etc. Ideally, the product owner should be involved in the initial definition of the systems requirements

specification (SRS), which is part of phase 4 and 5 (see Fig. 4.7) to ensure thorough understanding and ownership.

- **The Scrum team**: The Scrum team is the group of developers, testers and others that design, develop and document the solution—guided by the priorities and feedback from the product owner. Ideally, the team should be stable over time to establish and maintain team coherence and competency, but it is possible to make changes in the team or add specific expertise when needed. We have observed that it is beneficial that team members have some experience and understanding about safety; this enables awareness and an understanding or tolerance for added activities, beyond software engineering alone.

In addition to the traditional Scrum roles, there are six additionally safety-related roles which are needed to meet the requirements of IEC61508-3:2010:

- **Quality assurer (QA)**: The main responsibility of the QA is to ensure that all software quality-assurance tasks are done *throughout* the development process by those that are given the responsibility (see Chap. 7.6). In cases where issues are identified, the QA will ensure that corrective actions are taken as soon as possible. Given the size and complexity of the development, the QA-role may be taken by the Scrum master or it may be a dedicated person who could serve several teams, or the QA role may be shared on a rotational basis by, for example, some of the developers, given that they have the proper training. The QA must check that the developers in the Scrum team follow all safety plans, including the safety validation plan. In addition, it is important to check that the developers update the design if needed.
- **Independent Tester (s):** The independent tester is a specialized tester, not being a member of the team (as some assessors would object to having the developers test their own code, although this is not specifically stated in the standard). The independent tester may be part of a test department or a free resource from another team or project. Depending on the size of the development, it may be several independent testers. As the team members (developers) themselves take care of unit-testing (see Chap. 8.3.2) or focus specially on testing that the assessor allows to be done by the team itself, the independent tester may be responsible of higher-level tests, such as module tests (ref. IEC 61508-3:2010, section 7.47) and integration tests (ref. IEC 61508-3:2010, section 7.48). The IEC 61508-1:2010 has a detailed set of requirements for which tests could be done by an independent person, independent department or an independent organization. The choice will depend on the consequences and the SIL. See IEC 61508-1:2010, chapters 8.2.16–8.2.19 for more details. See also Chap. 8.3 (this book) for more details on testing.
- **RAMS Engineer**: The RAMS engineer is part of the alongside engineering team. He is thus indirectly involved in the SafeScrum® process and will receive evidence on proof of compliance with the standard from the team, alternatively also by having direct access to, for example, code, documentation, the product backlog and the sprint backlog. This role is responsible for the reliability, availability, maintainability and safety (RAMS) qualities of the system. In SafeScrum®, we focus primarily on safety. The RAMS engineer is responsible

of verifying that all safety requirements are fulfilled or that there are reasonable reasons for any avoidance. The RAMS engineer normally facilitates the communication with the assessor and is the central resource on safety for the team, the Scrum master, the QA, and the product owner. This role should be taken by someone with extensive knowledge on safety and the safety requirements, such as a safety expert. The RAMS engineer takes part in sprint planning and the sprint review and in any type of discussions or clarifications that are needed to evaluate the meaning of safety requirements and how they are met by the solution that is being developed. The RAMS Engineer functions as the liaison between the SafeScrum® software development process (Chap. 7.1–7.3) and the alongside engineering team activities, which may involve others external to the SafeScrum® team (Chap. 8.4). The RAMS engineer is also responsible for updating the agile hazard log and the safety case.

- **The alongside engineering team:**

  Alongside engineering is a collective name for a set of SafeScrum® activities that are done outside the sprints but mostly synchronized with these. The reason for synchronization is that some of the activities performed by the alongside engineering team are support activities for the sprint team. In addition, the alongside engineering team is responsible for all project activities that require safety and risk analyses competence. This includes but is not limited to:

  – Writing the safety plan.
  – Writing the plans for verification and validation.
  – Performing safety and risk analysis, both at the start of the project and each time there is a significant change to one or more requirements or to the system's operating environment.
  – Writing the initial agile hazard log, based on the initial safety and risk analysis and updating this document whenever there are significant changes to one or more requirements or to the system's operating environment.
  – Writing and maintaining the agile safety case. This includes checking that the development process is compliant with the standard.
  – Performing safety validation at the end of each sprint. Thus, the RAMS engineer is part of the alongside engineering team.

  In addition to the safety activities, the alongside engineering team is also responsible for writing the documents that can be written upfront, since they will not change during development. In addition, they will also write the first version of the system's documentation.

- **Coordination of SafeScrum® and alongside engineering results**—The SafeScrum® team produces documented code and corresponding unit tests based on information found in the sprint backlog—originally from the product backlog.

  The alongside engineering team does everything related to safety, such as risk and safety analysis, safety, V&V (Validation and Verification) planning and creating the initial hazard log and the initial agile safety case. See Fig. 6.2 for an overview. Due to the volatility of all plans, analysis and code stemming from

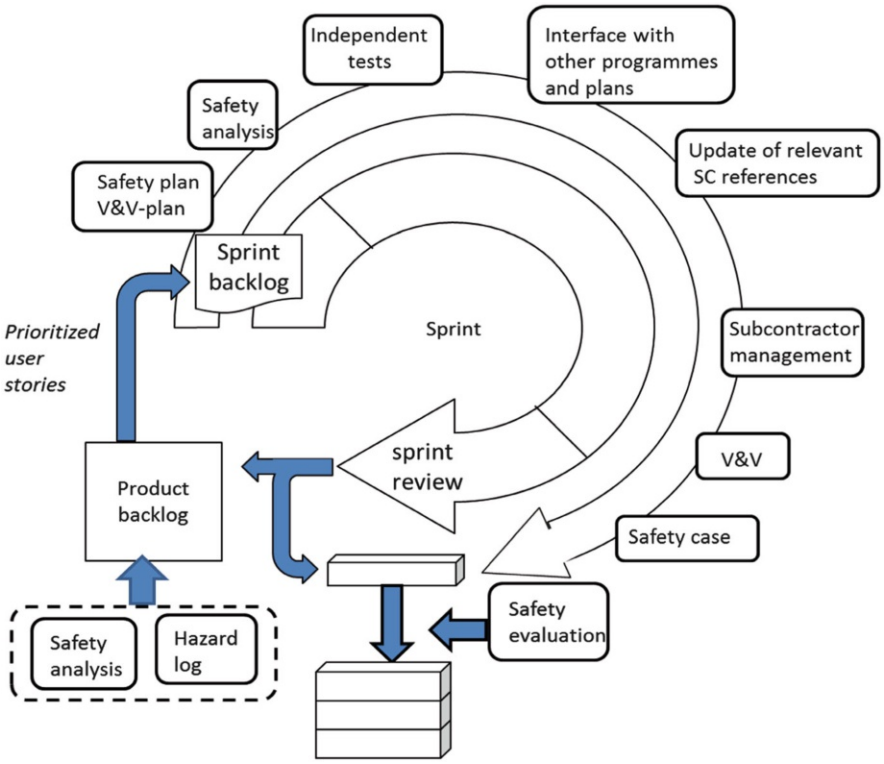

Fig. 6.2 The alongside engineering team activities

an agile development method, coordination is important in several cases. The following is a short summary:

- When a requirement is changed or added, we need to redo the safety and risk analysis of the impact that the change will have on the system's behaviour in the specified operational environment. In addition, we may need to update the hazard log, the agile safety case, the safety plan and the V&V plan. All this should be done as quickly as possible in order to have a correct picture of related hazards and the status of the safety case.
- When a development sprint is finished, we need to do a safety validation. This is the responsibility of the alongside engineering team and is done by the RAMS engineer. If there is a need for independent testing—for example, of a functional software unit—this is also part of the alongside engineering team's responsibility.

- **Independent safety assessor**: The external assessor is per definition and explicitly not a part of the development project but is indirectly involved in SafeScrum® and will receive documentation on proof of compliance with the standard from the team, via the RAMS engineer. The standard does not define the format of

Table 6.1 Assessor independence versus SIL

	Safety Integrity Level—SIL			
Minimum Level of Independency	1	2	3	4
Independent person	X	X1		
Independent department		X2	X1	
Independent organization			X2	X

documentation and this has to be agreed with the assessor. Safety case may be a useful format for documentation [2]. The assessor is responsible for assessing that all requirements in the standard are fulfilled. There are several ways to organize the collaboration with the assessor, but the basic principle is to establish frequent interaction from the start. The required degree of assessor independence can be found in Table 6.1.

- X: The level of independence specified is the minimum for the specified consequence or safety integrity level/systematic capability. If a lower level of independence is adopted, then the rationale for using it shall be detailed.
- X1 and X2: Factors that will make X2 more appropriate than X1 are:
  - Lack of previous experience with a similar design
  - Greater degree of complexity
  - Greater degree of novelty of design or technology
- **Project manager:** Although not a defined role in Scrum, most large projects are related to a higher-level stage gate process where a project manager (or similar role) is responsible for the coordination with other parts of the organization and with other development projects. A project manager may be seen as a link between the Scrum master and the team, and higher organization levels. Furthermore, a project manager holds the responsibility of the total system, including both hardware and software development. For more on project managers and the stage gate model, see Chap. 4.6.7.

6.4 Fundamental SafeScrum® Concepts

In order to establish full traceability from requirements to code in the SafeScrum® development process, we need to establish a set of basic concepts and how they are related—Fig. 6.3 gives an overview.

- **SRS**: The systems requirements specification contains all requirements and is the result of the initial phases according to the IEC 61508:2010 life cycle.
- **Requirements**: Requirements in the SRS may be either safety or functional requirements.

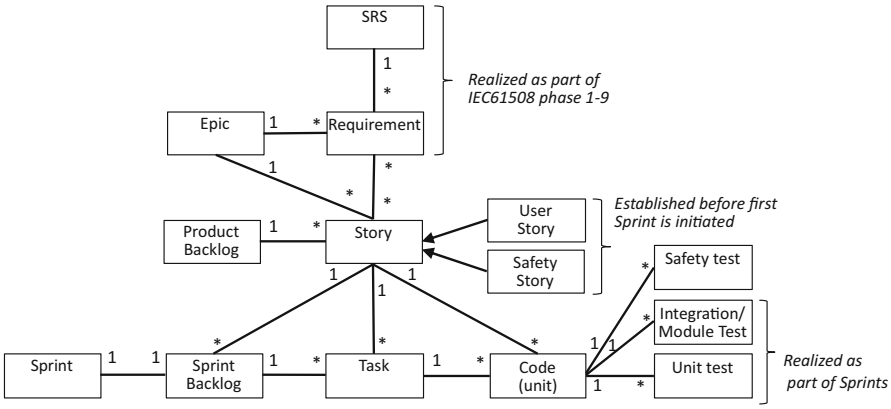

Fig. 6.3 SafeScrum® concepts and traceability

- **Story**: A story is a description of something the system should do, and may be either a user story (describing functional requirements) or a safety story (describing safety requirements). A story is often described in prose and non-technical terms and may be supported by additional useful information and references.
- **Task**: A task is a detailed work description, typically at a level that one developer can resolve. Tasks are typically defined when a story is broken down in a sprint planning meeting. When a story describes *what* to implement, a task describes *how* to implement it.
- **Epic**: An epic is a higher-order description of a large part of the system, typically covering multiple stories. It may be used to provide a better structure and overview of the stories and how they relate to each other. An epic typically contains or relates to several stories.
- **Product Backlog**: The product backlog is the container of all stories defining the systems requirements. It may either be composed of two backlogs, one for safety stories and one for functional stories or it may be one physical backlog where the stories are marked whether they are safety or functional stories.
- **Sprint Backlog**: A sprint backlog is a list of tasks that are selected to be resolved in a sprint.
- **Sprint**: A sprint is the fixed work period, typically 2–3 weeks, where the team works through the sprint backlog (and the defined tasks) to produce code or other artefacts that are needed to resolve the stories related to the tasks in the sprint backlog. Initially, longer sprints may be needed to settle the process, but these should be made shorter when the team gets used to the routine. The sprint length may be discussed in retrospectives in order to find the best pace for the team.
- **Code Unit**: A code unit can be a function, a method, or similar, which is the result of a task. A task is typically related to several code units.
- **Unit Test**: A unit test is a low/mid-level code-near test consisting of a set of assertions testing the interface of the code unit. Unit tests are typically managed by a unit test framework.

Integration and Module Tests In addition to unit tests (which only test low-level code), integration and module tests are also needed—see IEC 61508-3:2010, section 7.4.7 (module testing) and section 7.4.8 (software integration testing). As stated in the standard for integration and module tests:

"*This does not imply testing of all input combinations, nor of all output combinations. Testing all equivalence classes or structure-based testing may be sufficient. Boundary value analysis or control flow analysis may reduce the test cases to an acceptable number. Analysable programs make the requirements easier to fulfil.*"

Each software module shall be verified as required by the software module test specification that was developed during software system design. Software integration tests shall be specified during the design and development phase.

These tests are the responsibility of dedicated testers. Testers can either be members of the team, and/or be specialized independent testers.

For both module testing and integration testing, it is important to keep the traceability information up-to-date. This information is needed if we, during one of the sprints, change a module specification. Such changes require an analysis of the tests to see if one or more of them need to be changed.

System Safety Tests The objective of the system test is to ensure that the integrated system complies with the software safety requirements specification for the required safety integrity level. If compliance with the requirements for safety-related software has already been established, the validation need not be repeated. The validation activities shall be carried out as specified in the validation plan for software aspects of system safety. Depending on the nature of the software development, responsibility for conformance with the standard can rest with multiple parties. In SafeScrum® we recommend that this is handled by the alongside engineering team—see Chap. 6.3. The division of responsibility shall be documented during safety planning and accepted by the assessor. For more on safety testing, see Chap. 8.3.5.

6.5 Preparing a SafeScrum® Development Project

6.5.1 Create Initial Documentation and Plans

The software development process is based on and guided by several mandatory documentation and plans that provide important input to the software development process and other related processes like the overall project management (including hardware design and development, etc.), and that have to be established in advance according to the requirements in IEC 61508:2010. See Fig. 6.1 for some documents and plans of specific relevance to the SafeScrum® process.

SRS (System Requirements Specification)
The SRS is the key source of functional and safety requirements and the basis for setting up the initial product backlog that is populated with functional- and

safety-stories prior to the first sprint (see Chap. 6.5.2). The initial requirements are stated by the customer and are created based on his or her current understanding of the system and their needs. The following are examples of real customer requirements:

1. The system shall be provided with an emergency stop system.
2. The emergency stop system shall act on all motors.
3. The emergency stop function shall act on all hazards of the entire application.
4. The motor shall be fitted with a braking device to prevent the load from falling.
5. The braking devices shall be operated in such a way that the stopping time is as short as possible.

The SRS may contain:

– An overview section or a system overview including the solution, network topologies, integrity aspects, etc., the users of the system, constraints, assumptions and dependencies, and design guidelines. For an IEEE definition of design, see Chap. 4.1.
– The product (functional) requirements.
– The functional safety requirements and main safety concepts of the solution.
– The systems operational conditions.
– Operation of the system.
– Fault handling.
– External interfaces.
– Life cycle requirements.
– Information about environment, health and safety.
– Additional information, specific to the system being built.

Agile Safety Plan

The IEC 61508:2010 standard neither defines nor requires a safety plan, but it will nevertheless be a valuable document to support both the Scrum team and the alongside engineering team (see Fig. 6.4) in order to ensure a sufficient safety focus, hence the concept of an agile safety plan [3]. We borrow a definition of a safety plan from the EN 50126-1:1999 (3.39) railway standard:

"*A documented set of time scheduled activities, resources and events serving to implement the organizational structure, responsibilities, procedures, activities, capabilities and resources that together ensure that an item will satisfy given safety requirements relevant to a given contract or project.*"

The following list is an example of a generic safety plan, based on IEC 61508-1:2010:

- Develop understanding of the EUC and its operating environment.
- Specify the scope of the hazard and risk analysis, including system boundaries.
- Identify the hazards, hazardous situations and harmful events relating to the EUC.
- Develop a specification for the overall safety requirements.
- Allocate safety functions to the designated E/E/PE safety-related systems and other risk reduction measures—for example, barriers.

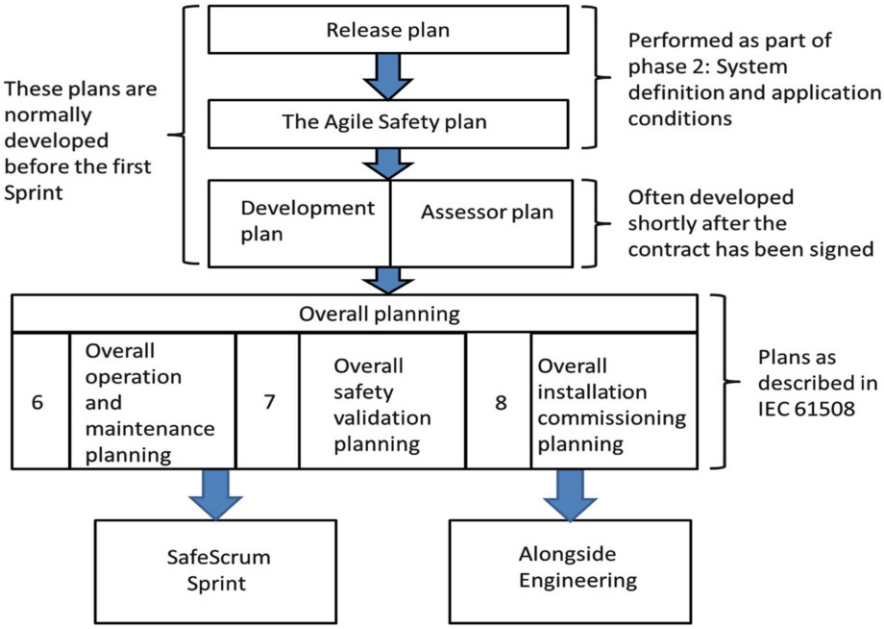

Fig. 6.4 From release plan, to high-level safety plan to the sprint planning as part of the "Overall safety lifecycle". The figure is based on the IEC 61508:2010 safety life cycle, EN 50126 safety life cycle and SafeScrum®

- Develop a plan for operating and maintaining the E/E/PE safety-related systems.
- Develop a plan to facilitate the overall safety validation of the E/E/PE safety-related systems.
- Develop a plan for the installation to ensure that the required functional safety is achieved.
- Specify the requirements for each E/E/PE safety-related system.
- Create safety-related systems conforming to the specification for the E/E/PE system safety requirements.
- Create risk reduction measures to meet the safety function requirements and safety integrity requirements.
- Make an installation plan for the E/E/PE safety-related systems.
- Validate that the E/E/PE safety-related systems meet the specification for the overall safety requirements.

In SafeScrum®, it is normally the RAMS engineer that is responsible for the Safety plan, which may be used as an important guide during SafeScrum® to ensure that all activities related to evaluation and decisions regarding safety are being done according to the original intent and plan. This is partly done by inserting the necessary activities into the product backlog and is followed up by the RAMS engineer and the QA responsible.

An agile safety plan helps the project manager (responsible for the total system, e.g. including hardware), the Scrum master (responsible for the software

development) and the RAMS engineer to track project tasks to a budget over time and it allows the Scrum master to keep management informed of progress. The agile safety plan is normally developed with contributions either from the project manager or the RAMS engineer depending on the project. A high-level version of a plan is management-oriented and includes an overview of how to satisfy the relevant safety regulations and standards, including safety plan requirements, for example, using the requirements for a safety plan as given in EN 50126-1:1999 section 6.2.3.4. Together, the agile safety plan, the high-level safety plan and the sprint planning constitutes the main agile plans.

While the agile safety plan should be established in phase 2 according to EN 50126, the detailed planning is performed in three separate activities as parts of work done in the phases 6 "Overall operation and maintenance planning", 7 "Overall safety validation planning" and 8 "Overall installation and commissioning planning" of IEC 61508:2010. Note that these phases are outside SafeScrum®.

Managers generally are concerned with approving a project before its initiation and then tracking it at the executive or program management level, for example, using a gate approach or similar, while the assessor is concerned with how the plan fits the assessment plan and concrete requirements for a safety plan. An important topic in the high-level project plan is the expected outcome. A project manager will explain in writing the purpose of a project and highlight the expected benefits. The assessor expects information related to, for example, audits, deliverables like V&V (Validation & Verification) reports and safety cases or similar documents. This is thus a project manager problem and does not involve SafeScrum®. For more on project managers and the stage gate model, see Chap. 4.6.7.

The Scrum master role and related sprint roles should be mentioned as part of the EN 50126-1:1999 section 6.2.3.4: "d) details of roles, responsibilities, competencies and relationships of bodies undertaking tasks within the lifecycle" requirement. A high-level plan will include later reviews by management. Management will expect to see interim deliverables or accomplishments, for example, milestones. Gate reviews are designed to allow management to decide whether to terminate a project, adjust the resources needed or allow it to continue. The gate reviews will be scheduled into the high-level plan.

The project manager, the product owner and the Scrum master are responsible for writing the delivery plan. This plan normally includes a time estimate. Assuming that the project manager will deliver something of value, people will be awaiting its delivery. Having an estimate of the delivery date allows the recipients of the projects deliverables to plan ahead for putting the deliverable to use. The plan should be kept up-to-date to communicate any major changes that may affect other roles and management.

Figure 6.4 below shows the links between the agile safety plan, high-level safety plan and the sprint planning. The "overall planning" is based on the IEC 61508:2010 safety life cycle as that life cycle presents the planning better than the EN 50126-1:1999 safety life cycle.

System Design
First and foremost, this is the high-level design—defining the system's architecture (outside SafeScrum®), the software system's design—see Chap. 4.1—and its main components. If not part of the SRS, the system design describes all subsystems and components of the total system, including the software architecture design. The designer(s) are responsible for the system design and may be part of both the sprint team and the alongside engineering team, depending on the project. There are no detailed guidelines in the safety standard on how to define the software architecture design, but generally it should describe how software components are separated, related or integrated to other parts of the system, how the software potentially is structured into subsystems, and any interfaces in between.

The system design has to balance between the right amount of detail upfront to avoid expensive changes—the architecture level, and room for being flexible when the software is being created—the detailed design level. Normally, the domain and type of application indicates typical high-level design or architectural patterns. See Chap. 4.6.3 for more details on system design and software architecture. To ensure a system design that is stable and well understood by the team and associated roles, it may be a good practice to include representatives of these roles in the development of the system design, as it will influence the rest of the software development process.

The level of detail will vary from case to case, but the system design should be as well defined as possible at the starting point to guide the software development. The system design should be stable without major changes later on as this may affect the functional safety and hence create a need to redo initial safety analyses. The system design guides the detailed design and the development of code in the sprints, meaning that any design decision needs to comply with the system design, which then becomes both a guiding and limiting factor.

Figure 6.4 refers to a set of plans which can be found in Chap. 4.5:

- Release plan—when do we plan to release which functionality?
- The agile safety plan.
- General validation plan—have we delivered functionality and quality as agreed?
- Assessor plan—what will the assessor check and when?
- Overall operation and maintenance plan.
- Overall safety validation plan.
- Overall installation and commissioning plan.

6.5.2 Creating the Initial Product Backlog

The initial backlog, which is the starting point for the first sprint, is created based on the system requirement specification (SRS), and is done as one of several preparations before the development sprints initiates. Some of the SRS requirements may be quite general—for example, the system shall be developed according to

IEC 61508:2010. The initial backlog contains the current best understanding of the requirements prior to development and will be refined throughout development based on feedback and experience with the solution under development, and a consequently increasing understanding of the requirements. Normally, functional stories may later be refined with more detail and precision while safety stories are likely to be more stable.

Defining the initial backlog may be a considerable task, depending on the size, complexity and clarity of the SRS. There is not necessarily a 1:1 relationship between the SRS and the stories in the backlog; defining user and safety stories based on the SRS, supported by the system design and other documentation, involves making several design decisions. This process should involve those responsible for the SRS that have the overall understanding of the system—first and foremost the product owner, who will be responsible of maintaining the product backlog—and the RAMS engineer who will ensure that the system becomes a safe system. In addition, we might also involve the team (or at least a representative) which is going to develop the software. In addition, it should involve the Scrum master, who will support the team and enable an efficient process. In cases where the RAMS engineer finds legislation and the standard unclear when a story is defined, or where a wrongful decision may have large consequences, the assessor should be consulted. Defining the product backlog can be seen as detailing the SRS and as a good way to learn about and understand the requirements. As traceability is essential, each story needs to refer to the requirement(s) in the backlog that is realized, for example, by reference to a unique ID in the SRS.

When a project is initiated, we insert both user stories and safety stories into the product backlog. Usually, the user stories come from the customer, while the safety stories may come from the customer, from a generic safety standard such as IEC 61508:2010, from the safety analyses and from the applicable domain standards. The SRS should be updated when a requirement is added to or removed from the product backlog. If we just change the interpretation of a requirement we have to decide whether it is a radical change—update both the SRS and the product backlog—or just a minor adjustment—only update the product backlog. See also Chap. 6.5.1.

One of the ideas of SafeScrum® is to operate with two backlogs; one containing user stories; mapping functional requirements, and one containing safety stories, mapping safety requirements. This separation is done because safety stories are more stable than user stories, which may be changed or refined during the course of a development process. As an alternative to having two physical backlogs, stories that are considered safety stories may simply be tagged. This is, however, a logical separation and stories may be gathered in the same physical backlog as long as the type of story is clearly marked.

Safety stories and user stories should be linked, meaning that user stories should refer uniquely to related safety stories—that is, safety stories that are present due to the requirements imposed by the user story—and vice versa. This is needed for the developers to evaluate how implementation may be guided or restricted by safety stories. For example, if a functional story is to be implemented, this link will inform the team about any related safety requirements that must be met through the implementation. This is important information that will affect *how* the functional

Table 6.2 A typical product backlog story description

ID	A unique and stable ID or reference number for the story.
Name	A descriptive name of the story.
Description	A description (reflecting the related requirement(s) from the SRS). Typically textual, but may also be supported by models or illustrations, and links to further information.
SRS_ref	A unique and stable reference(s) to related requirement(s) in the SRS that are covered by the story. Potentially, one story may implement multiple SRS requirements, and—one SRS requirement may be implemented in total by multiple stories.
Type	Story type; functional or safety. (Not needed in case of two product backlogs, but in that case, stories need to link each other to maintain the relationship).
Importance	Defined by the product owner as a value within a scale, alternatively an order of stories.
Tasks	Optionally, early ideas on implementation tasks that are needed to realize the story. These will be updated in the sprint planning meeting when the story is selected for implementation in a sprint and tasks are added to the sprint backlog.
Risk	Estimated level of risk for the story (High/Medium/Low). (How hard it is to implement). *Risk is used as part of the QA, see 7.2.3).*
Complexity	Estimated level of complexity for the story (High/Medium/Low). *Complexity is used as part of the QA, see 7.2.3).*
Estimate/Story points	An estimate of work (can be a number of size of the story or ideal work hours) needed to realize the story. This is just to give the team a track record on how much work they are able to do within a sprint.
Demo	A description of how to demonstrate the solution. For safety stories, this may be an assessment. For user stories, this may be a test or a reference to a test plan (not necessary with a demo for each story).
Definition-of-done	A Definition of what needs to be in place/approved for the story to be considered done *within the scope of the sprint*. Ideally, the team should be able to achieve this within the time and resources of one sprint. To align SafeScrum® with regulations and safety standards, Done-ness criteria have also to address: • The safety engineering activities that must be completed during and after the last sprint. This includes V&V activities and information, and documents to be produced including delivery to the relevant parties involved. • Authorization and the required documentation in that context for some domains like the railway domain.
Notes	Additional information and potential links to other documentation or plans that are useful for the team when resolving the story.

story will be implemented. In order to maintain the backlog properly, it is highly recommended to use a dedicated tool for this. This may be done by using a tool like Jira, which has basic functionality for backlog management or more specialized tools such as RMsis which adds support for managing traceability, change management, etc. Spreadsheets or similar tools may also be used but would probably require extensive manual work and also a risk of inconsistency.

A story in the product backlog may be defined by the information shown in Table 6.2.

As stories may change over time, we need to use a tool to establish traceability and consistency of these changes.

A system may encompass a large number of stories and it may become a challenge to maintain a full overview of the system. Hence, it may be useful to adopt the concept of epics from Scrum and agile development. Epics are in many respects stories, which are too large to be completed in one sprint and are a higher-level description of, for example, the main sections or functions of the system. Epics are useful to capture and document a higher level understating of the system. Each epic will later be broken down into many stories and are usually implemented across several sprints.

6.5.3 User and Safety Stories

User stories may be changed or added throughout an agile project. As a lot of things in agile development, creating user stories is also part of achieving efficient communication in the project. Thus, who writes the user story is far less important than who is involved in discussing it later. Most of the stories in the product backlog are user stories. Safety stories can be considered as a special case of user stories. Both user stories and safety stories will bring several benefits to the project. The most important ones are:

- It will create discussions about how to realize the stories, both when writing them, during the sprint backlog refinement discussions and when they are to be implemented. Discussions lead to communication and a common mental model of how to realize the story, which will improve system quality.
- We will discuss how to realize the story when it is selected for implementation. Thus, our decision will be based on more information than if we had decided how to realize it at the start of the project.

A user story will have one of the following layouts:

- **As a** <user role>, **I want to** <achieve some goal> **so I can** <reason>
- **As a** <type of user>, **I want to** <perform some task> **so that I can** <reach some goal>

It is recommended to attach one or more acceptance tests to each user story. This will be useful both as an extra piece of information for the interpretation of the user story and as a part of the user story acceptance test. See next section for details.

Safety stories may also be changed or added throughout an agile project but will, in the general case, be more stable than the user stories. Just like user stories, the safety stories will improve the communication. In addition, the discussions created when dealing with the safety stories will help in creating a safety culture in the team. Since the safety stories are about "what" and not about "how", they are a good starting point for discussions with the assessor or the RAMS engineer, who will be responsible for the final safety validation. For example, will the assessor accept that we solve this safety story by implementing a specified barrier?

A safety story will have one of the following two layouts:

To satisfy <a safety standard requirement> **the system must** <achieve or avoid something>

To keep <function> **safe, the system must** <achieve or avoid something>

Linking safety needs to software design is already required by the IEC 61508-3:2010. In SafeScrum®, this is done by having two backlogs or, alternatively, one backlog having different tags on safety stories and user stories, and links between functional and safety stories (see Fig. 6.3).

Even though the product backlog is in some senses the equivalent of a requirements document in a non-agile project, it is important to bear in mind that a user story or safety story is not finished before it is discussed both within the team and with the product owner.

The introduction of safety stories into agile development will enable us to involve developers, assessors and customers in realizing the system's safety requirements, help to build a safety culture and base our safety-related decisions on all currently available information. Safety stories are really safety requirements. They are, however, inserted to avoid or reduce hazards. To make the story-arsenal complete, we have thus added hazard stories.

Hazard stories are not intended as safety requirements but a description of an identified danger and stems from the work of K. Łukasiewicz at the University of Gdansk [1]. The idea has much in common with the original user stories and the safety stories.

Hazard stories are written based on epics—the big picture—and the user stories—what the system will do for the users and why it should be done. For SafeScrum® we may also use hazard stories as input to the safety stories. Just as safety stories they will be inserted into the safety backlog. If <accident condition> in a hazard story corresponds to a <achieve or avoid something> in a safety story, the hazard story gives a safety requirement, just as a safety story. If not, we need to write a safety story which will help us avoid the accident condition (see also Fig. 6.5).

The process used to generate the hazard stories is straight forward: use the format shown below, get all relevant persons together and brainstorm based on the information available. The format is as follows:

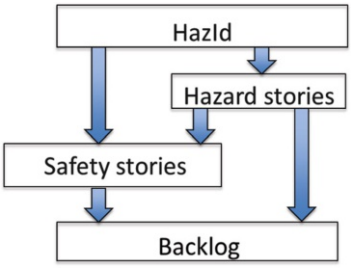

Fig. 6.5 Hazard stories and safety stories

As a result of <cause> <cause event> **which will lead to** <accident event> [**if** <accident condition>]

This format contains the same two information items as a HazId table—"failure condition = cause event" and "effect of failure = accident event". In addition, it contains information on the cause and the accident condition. This is important information when we have to decide how we will reduce or remove the hazard. The cause information might help us to remove or reduce the root cause of the accident while the accident condition may identify ways to make the system more robust by identifying possible barriers. Since the hazard stories are based on elements already used in agile development, it fits well with SafeScrum®. The following is a simple example:

As a result of <user light-heartedness> <phone may be lost> **which will lead to** <possible unauthorized access to the app> **if** <the app isn't secured, for example, with a pin code>

Hazard stories support discussions and decisions—they do not lead directly to features or functionality.

6.5.4 Setting Up the Team and Facilities

The team needs to have members with the right experience and competency with respect to the system under development, such as skills in chosen technologies, tools, languages, etc. There is no golden rule on the number of team members, but somewhere between 4 and 7 seems to be quite common. However, there may be reasons for less or even more—this has to be considered in each case. The goal is, however, to have a team that together have the competency that is needed and that is small enough to be able to collaborate. In addition, team members should go well together at the personal level and previous collaborative experience is thus positive. In addition, for safety-oriented projects, a general understanding of safety is highly valuable as a SafeScrum® team continuously needs to reflect on how their decisions may impact safety.

The team should ideally be co-located. This enables frequent and easy interaction, both for frequent meetings like the daily stand-up (see Chap. 7.5), and for ad hoc discussions during the workday. If the team is distributed, it is absolutely necessary to use video conferencing and shared desktop facilities. It is also useful to have a dedicated room where information of common interest is displayed, for example, a board displaying information about the sprints, backlogs, burndowns and a board displaying the system architecture, etc. This can be done by the use of whiteboards or by having large screens displaying information from a workflow system like Jira or similar.

6.6 SafeScrum® Key Process Elements

SafeScrum® inherits the key elements from Scrum. We have, however, made some additions (1) in order to make it compliant with IEC 61508-3:2010—for example, trace and the RAMS engineer—and (2) due to feedback from one of our industrial partners—for example, the added QA responsible role.

The three key processes involving the defined roles (see Chap. 6.2) are the sprint planning meeting, the sprint workflow (with an added explicit QA-role), and the sprint review meeting. Figure 6.6 illustrates the workflow through these activities, which iterates several times (the sprints), until all stories in the product backlog are done. Details are described in Chaps. 7.1–7.3. In addition to these three key processes, SafeScrum® also includes some process elements that support collaboration and process improvement. These processes are sprint retrospectives, which enable dynamic improvement of the processes (Chap. 7.4), the daily stand-up meeting, to uncover and resolve potential problems (Chap. 7.5), and change impact analysis to assess any potential safety impact related to major changes (Chap. 8.2).

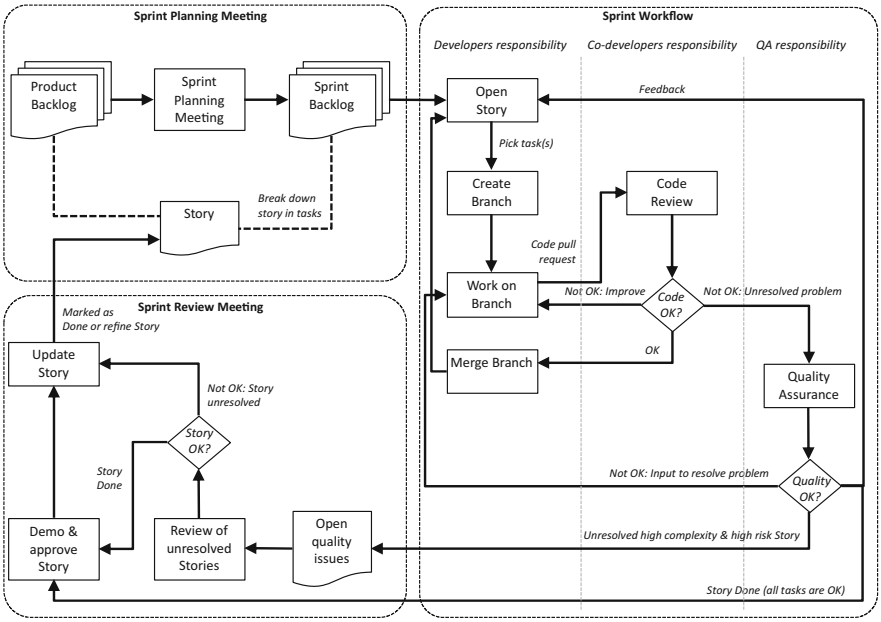

Fig. 6.6 SafeScrum® key process elements

References

1. Łukasiewicz, K. (2017) *Method of selecting programming practices for the safety critical software development projects – A case study*. Technical report no. 02/2017. Gdańsk University of Technology.
2. Myklebust, T., & Stålhane, T. (2018). *The agile safety case*. Berlin: Springer.
3. Myklebust, T., Stålhane, T., & Lyngby, N. (2016). *The agile safety plan.* PSAM13.

Chapter 7
The SafeScrum® Process: Activities

This chapter present the main Scrum activities, re-casted into SafeScrum®. We discuss important activities such as:

- *Sprint planning, workflow, review meetings and retrospectives.*
- *The daily stand-ups.*
- *Backlog refinement—an important part of Scrum.*
- *Explicit quality assurance—a necessary addendum to Scrum.*

7.1 Sprint Planning Meeting

Each sprint starts with a planning meeting where the Scrum master, the product owner, the QA and the team are present. If needed to clarify safety requirements and decisions (e.g. when detailing tasks), the RAMS engineer should also be included. Experience shows that a planning meeting may take 1–3 h. However, this may vary with the size and collective experience of the team, and the clarity of the stories. Lengthy meetings are often a sign that stories are not defined clearly enough and that they maybe should be refined—see Sect. 7.6 on backlog refinement.

The sprint planning meeting comes after the sprint review meeting of the preceding sprint with the aim to (1) define the goal of the next sprint, (2) decide who will work how much in the team, and (3) prioritize and select the stories to resolve and define more detailed work tasks.

The timing of the sprint planning meeting is flexible. It may be done right after having finished the review meeting for the previous sprint, when knowledge and results from the review is fresh in mind for the participants, typically the last half of a Friday. However, this may be exhaustive for some teams. Alternatively, it can be done on the first day for the new sprint, for example, on a Monday, giving the team time to rest and reflect.

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_7

7.1.1 Defining the Sprint Goal

The sprint goal is a short statement explaining why the planned sprint is being done and should be defined collaboratively by the product owner, the Scrum master and the team. A sprint goal is a high-level and short description that is a useful reminder of the purpose of the sprint, used to maintain focus when developers work on the details. It is also useful in cases where multiple teams work on the same product to keep each other informed. The sprint goal may be documented on a wiki-page or similar.

7.1.2 Clarifying Team and Commitment for the Sprint

Ideally, the team should be stable, meaning that the same persons should remain in the same team over time. This strengthens team cohesion, maintains the shared knowledge and makes planning easier. However, there may be reasons for variation, such as sick leaves, parental leaves, duties in other projects, training, or the fact that a specific developer is required for specialist tasks. The sprint planning meeting should thus also define the team for the upcoming sprint and consider whether the RAMS engineer is needed to clarify one or more decisions. It is also necessary to decide who should take the role as QA in case this is not a fixed role in the team.

7.1.3 Creating the Sprint Backlog

The product owner, the team and the Scrum master take part in defining the sprint backlog. Additionally, the RAMS engineer may also participate if we expect discussions regarding safety. The safety plan is also a helpful artefact to inform the project participants about safety issues. The product owner is responsible for selecting the top prioritized stories but will discuss this with other roles such as the RAMS engineer. All the stories have a defined priority and an initial estimate from the initial planning and from refinements, and the simplest procedure would be to select stories from the top until the estimates match the available resources in the team for the upcoming sprint.

Stories are broken down into tasks, which are short and precise work descriptions. This breakdown adds more detail and is actually a design process as the team creates ideas on *how* to realize the selected stories. In some cases it may be relevant to clarify the design choices with the RAMS engineer to be sure that the design does not conflict with related safety stories.

7.2 Sprint Workflow

7.2.1 Resolving Stories

After having finished the sprint planning meeting with the team and the sprint backlog has been defined with selected stories, the working part of the sprint starts. Developers open stories from the sprint backlog and break them down into workable tasks. Thereafter, they work on each task, which can be code development, bug fixes, creating documentation, testing ideas, establishing infrastructure, etc.—basically work that has to be done. Each developer uses a workflow management system, such as Jira, to pick a story and mark it as "Open" or "In Progress". For coding tasks, a branch is created in the code management system, for example, Git. New code is developed according to the principles of test-first development (see Chap. 8.3) and any changes to existing code needs to be supported by updates of the unit tests.

7.2.2 Peer Review of Code (Pull Request)

When the developer considers himself or herself to be finished with a branch, including passing the unit tests, they request a code review by another developer in the team, also known as a pull request. The version control system may be set up to enforce this review to be performed to avoid accidental neglects.

If the reviewer concludes that the code and its unit tests are OK, the QA is notified and checks against predefined quality parameters (see Sect. 7.2.3). If the QA finds an issue, he or she checks the defined risk and complexity of the story (part of each functional user story), and if both are set to 'low', the developer is notified and asked to improve the code. If, however, either risk or complexity is set as "medium" or "high", the story is added to a list of open quality issues to be reviewed as part of the next sprint review meeting. This is a precaution, added to ensure that any unclarities that have a medium/high risk and complexity are discussed by all, and that low risk and complexity issues can be resolved within the sprint.

Using a tool such as Bitbucket may ensure that this interaction, including comments and responses from the developer and the reviewer, can be documented and viewed in combination with the code itself. This is useful documentation, which establishes traceability of the low-level process, and may be useful for the assessor later on when the system is to be evaluated and certified. It may also be valuable documentation for later work on the code.

7.2.3 Quality Assurance of the Code

When the peer review is done; either by the reviewer approving the code or by the developer and the reviewer agreeing that they are not able to resolve the problem for

any reason, the QA is notified. The QA will check the code and the documentation and provide feedback to help the developer resolve any problem. The developer checks that the following quality parameters have been analysed and are within acceptable limits. See Sect. 7.6 for details. Other quality checks may also be added as needed:

- Peer review (pull request) comments
- Code metric values for new or changed code
- Documentation coverage
- Test coverage
- Requirements-task-code traceability

Most of these quality checks can be automated by tools, but we recommend that the QA makes sure that the analyses are done and that the outcome is acceptable.

If the QA finds that the quality is OK according to this list, the developer is notified and may check in the branch and eventually mark the story as done if all tasks are done. As mentioned above, if some quality issues are found in stories where either risk or complexity is set as medium or high, the quality issue is added to a list of open quality issues to be resolved in the sprint review meeting.

The purpose of the QA role and the extra quality check is to resolve issues within the sprint as far as possible and restrict the amount of issues that have to be discussed in the sprint review meeting.

The added code review, the QA within the sprint supported by tools, and the review of remaining quality issues in the sprint review constitutes three levels of quality control that can be documented and is an approach to avoid low-quality code, in particular for stories that are defined to have medium or high risk and complexity.

7.3 Sprint Review Meeting

The sprint review meeting ends the sprint and the results from the sprint are evaluated against the sprint goal and the stories that were selected for the sprint backlog in the sprint planning meeting. In SafeScrum®, the sprint review has two parts: (1) reviewing tasks with unresolved quality issues (from the open quality issue list)—see Sect. 7.2.3, and (2) reviewing or demonstrating resolved tasks from the sprint backlog. If the result is executable code, this will be done by running a demonstration—the standard Scrum approach—but if one or more sprint activities have produced or updated documents, it will be done by reviewing the documents or having a walk-through of the documents produced.

The first part of the sprint review requires the team, the Scrum master and the QA to participate. If needed, the RAMS engineer and other experts may also be included, for example, in case there will be discussions about safety implications. This part of the meeting will review and resolve all tasks that could not be resolved by the QA during the sprint (added to an "open issues" list, see Fig. 6.3). This may be stories where there are tasks that break defined code metrics, or tasks related to safety

requirements where the QA need to discuss possible safety impacts with the whole team, the product owner and the RAMS engineer. In some cases where the code doesn't meet the defined quality metrics, the sprint review meeting may decide to accept this and add an explanation for the exception. For example, if some code exceeds a defined complexity metric (e.g. STPAR—number of parameters) there may be a good reason for this, which should be documented.

In the first part of the sprint review, problems are discussed and the story is updated and put back into the product backlog to be resolved in a later sprint if the code needs further work. In that case, the story is updated with information that is needed to resolve it later on and avoid the problems or obstacles that were experienced in the recent sprint. If the sprint review meeting, however, finds the QA-deviation acceptable and the reasons for this are documented, the story may be marked as done.

The second part of the SafeScrum® sprint review corresponds to a typical sprint review meeting in Scrum where also the product owner participates. The intention is to evaluate the result against the requirements and expectations of the product owner, regarding the *functionality* of the system, stated as user stories. All stories and their solutions are presented and demonstrated to the product owner, which provides feedback to the team. Demonstration may be done, for example, by running code, showing a test report or simply displaying and explaining work that has been done and its results. Ideally, the story should describe how it should be demonstrated. The product owner approves stories marked as done. However, if the product owner is unsatisfied with the result, the story goes back to the product backlog to be resolved in a later sprint—preferably the next one to benefit from fresh-in-mind experience. In such cases, the story needs to be updated with new knowledge or details that are needed to resolve it. The sprint review may also cause the product owner to want something new, resulting in the definition of a new story or refining remaining stories in the product backlog.

Like the sprint planning meeting, the sprint review needs the team, the Scrum master and the product owner to participate. Other personnel may also participate as a means to spread knowledge about the development—both the solution and the process, for example to other teams. To meet the standards' requirements of traceability, all decisions and uncovered problems in the sprint review must be documented, for example in a workflow tool such as Jira Agile.

7.4 Sprint Retrospective

While the sprint review meeting evaluates the work *results*, the sprint retrospective meeting evaluates the work *process*. This meeting typically involves the Scrum master and the team, and the aim is to evaluate all routines, roles/responsibilities, tools, etc. The sprint review may be done after each sprint or whenever there is a need to evaluate the process. The meeting can be organized as a conversation to highlight problems and needs for improvements, or more formally as a post-mortem

analysis [1]. Either way, the goal is to identify actions to improve the process based on experience from previous sprints. The type of problems and corresponding improvement actions will vary greatly and may cover issues such as sprint length, team composition and competency, use of tools, office facilities and so on. Retrospectives are particularly relevant for teams that are applying SafeScrum® for the first time.

7.5 The Daily Stand-Up

The daily stand-up—also known as the daily Scrum or simply the stand-up meeting—is, as the name indicates, a daily meeting. Normally, it is done in the morning and should be kept short and relevant without detailed technical discussions. Alternatively, the stand-up may be done right before lunch to encourage a short meeting. One common way to achieve this is to have everybody stand up throughout the meeting to avoid lengthy and unfruitful conversations; 15 min should be enough time. There is no common rule for these meetings, but it is common to focus on three questions for all to answer:

1. What was done yesterday in order to meet the defined sprint goal? Take care that this does not develop into a traditional status meeting.
2. What will be done today?
3. Do I have any problems hindering my work?

Any problems that are found are discussed *after* the stand-up, and only by those that are needed, to avoid having the entire team spend time on discussions that are not relevant to them.

When developing a safety-critical system, it may be wise also to add a fourth question:

4. Do I see anything that may compromise safety?

This might also include adding new hazards to the hazard log—see Chap. 8.4.2. If the answer to the last question—question 4—is positive, we need some additional process. First, we need to close the daily stand-up meeting. Those who have the necessary competence stay for the safety meeting to discuss and resolve the safety issues. If this proves difficult, we should involve the RAMS engineer or, if this also fails, we should involve the assessor.

There is no need to record any minutes for any part of the meeting, the value of the daily stand-up is to keep everybody informed and quickly highlight any problems.

7.6 Backlog Refinement Meeting

One of the main outcomes of development in the sprints, besides software and related artefacts, is updated knowledge of the system, its design and its requirements; SafeScrum® is also a framework for learning and improvement. This means that the product backlog needs to be refined based on new, improved knowledge. This may be done as part of the sprint planning meeting, but this carries the risk that the meeting will become very detailed and time consuming. Thus, it may be better to organize separate refinement meetings when needed. This can be either before the sprint planning meeting or during the course of the sprint, for example, halfway through or when the need to refine the backlog is large enough. This gives the product owner time to resolve and check out any un-clarities before the next sprint planning meeting, which should focus on prioritization of stories to resolve, and not so much on detailed discussions related to unclear stories. The team, the Scrum master, the product owner and possibly the RAMS engineer should participate.

Functional requirements may influence system safety. Thus, the backlog refinement is important to get an understanding of how the functional requirements will influence the safety requirements. If in doubt regarding safety requirements related to legislation or standards, the assessor should be consulted as soon as possible. The intention of backlog refinement meetings is not to define new requirements but to improve the understanding of the existing requirements and as a result ensure that requirements are implemented correctly. In most cases, the backlog refinement process will not require SRS changes. If this, however, should be the case, a dedicated requirements meeting should be held (see Chap. 8.2 on change impact analysis).

7.7 Additional Quality Assurance

SafeScrum® may be seen as a development process with inherent and built-in quality assurance activities. Repeated evaluation of results in the sprint review meetings, daily stand-ups and peer reviewing of code are activities that evaluate and improve both the understanding of requirements and the code quality. In addition to this process, SafeScrum® will also strengthen quality assurance by explicitly assessing code metrics, source code documentation coverage and test coverage. Note that the standard does not have any requirements for or definition of source code documentation or documentation coverage (other than that which is needed). Thus, the project has to define this in their coding standard and get it accepted by the assessor.

7.7.1 Coding Standard and Quality Metrics

The project developing safety critical software must have a coding standard both according to the safety standard and in order to improve communication within the team. In addition, a coding standard will make it easier to perform code reviews and to maintain code written by others. The QA role will make use of the coding standard when checking code during the sprints.

IEC 61508:2010 requires that we have a coding standard—see IEC 61508-3:2010, tables A4 and B1. See also IEC 61508-7:2010, section C2.6.2 for some advice. A programming language coding-standard should:

- Specify good programming practice.
- Proscribe unsafe language features; constructions that should not be allowed or only allowed under specific, documented circumstances.
- Promote code understandability. This is important for, for example, code reviews and maintenance.
- Facilitate verification and testing.
- Specify procedures for source code documentation.

Where practicable, the following information shall be linked with the source code:

- Legal entity—for example, a company and authors.
- Description—what does this code do and how does it do it.
- Inputs and outputs—names, types and their meaning.
- Configuration management history—see IEC 61508-7:2010, section C.5.24

Some standards, such as IEC 61508:2010, want to eliminate or reduce the use of pointers, recursive code and such like. This does not mean that pointers, for example, are forbidden. What it means is that you should document where they are used and the reason why they are needed.

It is important to control code complexity. It is also a requirement from some safety standards. The method needed to do this can vary from an advanced metrics regime to a simple process where somebody assesses the code as OK or too complex, based on experience. Note that metrics cannot be used as predictors for anything—they are just useful indicators. Some of the metrics used in industry are Henry-Kafura's fan-in fan-out metrics and McCabe's cyclomatic value—v(G). There has been a lot of criticism levelled at McCabe's cyclomatic number. Even so, it is still used a lot in industry—not for prediction of error density or content but as an indicator for code complexity.

Besides the problem of choosing one or more metrics, we are also faced with the problem of choosing an action limit as IEC 61508:2010 does not define specific limits. We do not achieve complexity control by using McCabe's cyclomatic number if we do not at the same time define a limit for this number. We can use a rule such as: "If v(G) is greater than five, the developer shall either rewrite the code to reduce the value or write a short note explaining why the higher-than-normal v(G) is

permissible here". We may use the rules defined by others or use these rules as a starting point and modify them as we gain experience.

The module size metric is important for two reasons: it sets a limit to the number of code lines a developer has to simultaneously "keep in his head" and it will decide the lowest level for traceability. As an example, we will consider IEC 61508-7:2010, appendix C 2.9. We asked a representative from a European certification organization to give us a recommended size for subprograms and modules and got the following response:

- *"Subprogram sizes should be restricted to some specified value, typically, two to four screen sizes"*.

This gives a subprogram size of 200–400 lines of code.

- *"A software module should have a single well-defined task or function to fulfil"*. This definition allows for several interpretations. We recommend the size not to exceed 1000 LOC for modules in order to have clearly arranged and structured software architecture.

The same European certification organization does, however, add an important remark: *"In general we interpret a module as a set of code which fulfils a defined function; this makes also sense from a testing point of view (test specification level)*. Furthermore, *. . .for us it is more important to have a well-structured architecture with defined function modules than to insist on defined LOC restrictions"*.

Table 7.1 shows the metrics and the limits used by a company that uses the metrics tool PRQA. Note the text "Current limit". The limit for each metric is not set once and for all but may be changed as we gain new experience. STPTH, etc., are the tool's internal terms for the metrics. The PRQA tool, used by the company supplying the data shown below, uses a rather simplistic method for estimating the number of static paths—STPTH. Thus, it might be advisable to handle this value with care—it might be too big if the code contains one or more "SWITCH/CASE" statements. See the table below for the other parameters used. In addition, it is useful to discuss this approach with the assessor—at least the limits set for each metric (Table 7.1).

Note that the company involved uses Myer's metric instead of McCabe's complexity metric. McCabe's metric is the number of independent paths through the code, while Meyer's metric is the number of logical conditions. The two numbers will differ if one or more branching points contain compound logical expressions.

Table 7.1 Example of limits for metrics

Metric ID	Current limit
Number of static paths—STPTH	40
Number of parameters—STPAR	3
Function call count—STSUB	30
Max nesting of control structures—STMIF	5
Number of executable lines—STXLN	50
Number of maintainable lines—STLIN	70
Myer's value—computed (STMCC)	10

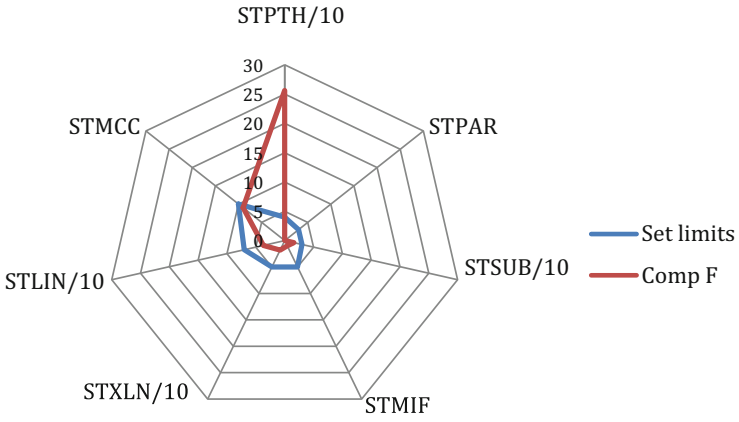

Fig. 7.1 Example of a radar plot for metrics

In order to simplify the work for those who shall check the metric values—for example, the QA role (see Sect. 7.2.3)—we recommend using a radar plot to show the metrics for each component together with the current recommended metrics limits. Using this plot, it is easy to see if the metrics are within recommended limits and if not, where the problem lies. In the plot example below, we see that it is the number of static paths that might be a problem. Some of the metric values (e.g. number of static paths and number of executable lines) are scaled down to make the plot more readable (Fig. 7.1).

As a final warning, we must bear in mind that a too high metric value is not a proof that something is wrong. It is just a warning, saying that we should have an extra look at this component.

There exist several coding standards, such as the GNU coding standard, NASA's 10 rules for developing safety-critical code, the MISRA coding standard and Science Infusion Software Engineering Process Group's "General Software Development Standard". Note that IEC 61508:2010 just requires the project to *have* a coding standard—it does not say which one. IEC 61508-3:2010, section 7.4.4.12 says: *"Programming languages for the development of all safety-related software shall be used according to a suitable programming language coding standard"*. Hence, the coding standard needs to be defined in each case and it may be useful to discuss this with the assessor at an early stage.

7.7.2 Code Documentation Coverage

New or changed code needs to be properly documented. This is valuable both for maintenance of the code, for review by the QA and for assessment. Inline documentation (e.g. as comments in or related to code) is preferable, but it may also be separated in a dedicated documentation system, referring back to code. Exida has

suggested the following definition of comments density: "*Relationship of the number of comments (outside of and within functions) to the number of statements*" [2].

By the end of the sprint, the QA will check the resulting code documentation coverage. Exida [2] suggests that the documentation coverage, as defined above, should be larger than 0.2. If it is found to be insufficient it should be defined as a task to be resolved in the next sprint.

7.7.3 Unit Test Coverage

First and foremost—unit as it is used in the software community is different from how some of the assessors use this term. We have found that some of the assessors used the term unit as synonymous with "functional unit" while most software developers use the term for a "chunk of code", like a function or method.

IEC 61508:2010 has no requirements for test coverage for unit testing. As a matter of fact, it does not mention the term "unit testing" at all. However, some of the assessors add their own requirement, for example, a minimum 95% test coverage at the unit-level. This is checked by the QA each time a developer makes a pull request. It may be done by a manual check of code and tests, but when the code is large or complex it may be supported by a code coverage tool like, forexample, Squish Coco or similar.

References

1. Birk, A., Dingsøyr, T., & Stålhane, T. (2002). Postmortem: Never leave a project without it. *IEEE Software, 19*(3), 43–45.
2. Moore, J. F. (2018). Software metrics. In *Exida explains Blog*. Exida.

Chapter 8
SafeScrum® Additional Elements

This chapter discuss SafeScrum® add-ons related to safety and IEC 61508:2010:

- *Traceability of requirements.*
- *Changes and change impact analysis.*
- *Testing.*
- *Safety engineering.*
- *How to manage releases in an agile context.*

8.1 Traceability

The notation next to the arrows in Fig. 8.1 refers to the relevant tables in IEC 61508-3:2010, appendix A. Strangely enough, there is no requirement for trace from design to SRS (Safety Requirement Specification) or from test specification back to design. This is, however, due to an editing mistake in edition 2 of the standard, creating inconsistency between the requirements for traceability in IEC 61508-3:2010 (Annex A) and the description given in IEC 61508-7:2010 (C.11). These traces will most likely be added in edition 3 of the standard.

System safety requirements are the collection of all safety requirements, whether they are related to software, hardware or wetware (humans).

In order to decide the level of trace—how far down into the system structure we need to go—we need to decide the granularity of the traces (Table 8.1). Since the standards are generic, they do not prescribe the required granularity. SafeScrum®, which aims to move all standards towards a goal-based approach, believes that each company should define their own granularities. The opinions differ between assessor companies. According to the diagram above, we need to have traces down

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_8

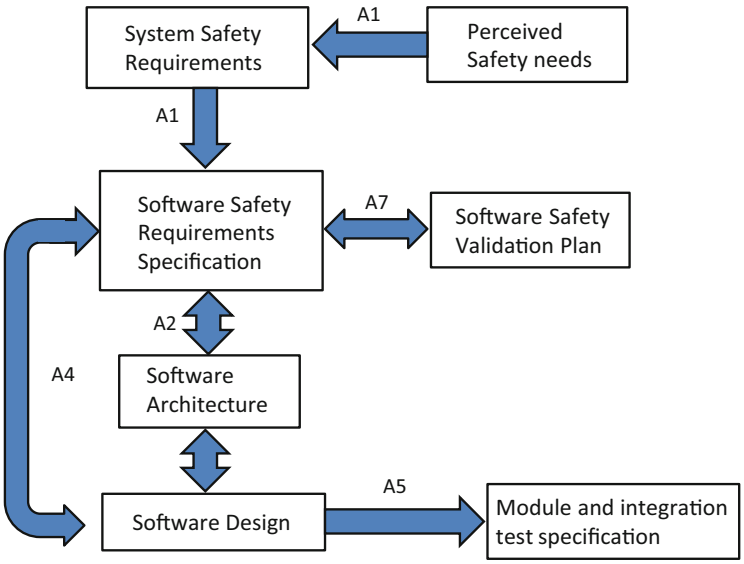

Fig. 8.1 Traces required by IEC 61508:2010

Table 8.1 Traceability links

SRS (requirement ID) ↓↑ product backlog (user or safety story)	For each story, annotate the unique ID in the SRS that the story is derived from. This can be as simple as a defined field in each story. Jira offers the opportunity to customize templates where this can be added.
Story ↓↑ Task	Each story is broken down into tasks which are detailed descriptions on how to realize the story. This may be managed and stored by the workflow system, e.g., Jira.
Task ↓↑ Code (unit)	The task-code traceability can be managed by using a code repository system that either offers this feature or by adding another tool that takes care of this relationship, e.g., codeBeamer. Tracing code at a unit (method or function) level is sufficient.
Code unit ↑↓ unit test	Unit-testing is supported by a unit-test framework, like NUnit, which also keeps track of the code-unit-test relationship.
Code unit ↑↓ System test	System tests should be covered by a dedicated test tool keeping track of which code unit is covered by which system test. There are, for example, several tools that integrate well with Jira.

to module level. IEC 61508-7:2010, appendix C.2.9 “Modular approach” has the following definition

> *“a software module should have a single well-defined task or function to fulfil”.*

In addition, Part 3 of the standard in section 3.3.5 defines a software module as a

> *"construct that consists of procedures and/or data declarations and that can also interact with other such constructs".*

At least one certification company has recommended a maximum module size to be 1000 lines of code.

The text "Ax" in Fig. 8.1 shows that the trace requirement is stated in annex Ax in IEC 61508-3:2010. All trace relations are highly recommended for SIL 3 and recommended for SIL 2. Note that that standard does not require traceability down to module level. This will, however, be changed in edition 3 of the standard. On the other hand, such a trace would be valuable for maintenance activities and change impact analysis.

SafeScrum®, supported by the use of tools, enables documentation of traceability by keeping track of references from requirements in the SRS, to defined stories in the product and sprint backlogs (and potentially to epics), to code units and to tests (see Fig. 6.3 for relationships). This trace information is maintained and stored by the collection of tools used for requirements, workflow, code, and test management. See Chap. 10 for more details on tool categories. However, we will only discuss tool categories—for example, process tools or tools for test and code analysis—not how to use any specific tool as there may be many alternatives and ways to combine tools. Any mention of a concrete tool are only examples.

8.2 Change Impact Analysis

8.2.1 Introduction

In all safety-critical software development, any change might compromise the existing system safety. Thus, we need to perform a change impact analysis to be sure that the system stays safe after the changes. In SafeScrum®, the need to consider a change impact analysis will arise in two cases—(1) changed or new requirements or (2) at the start of a sprint—the sprint refinement and planning meetings. The first case—changed or new requirements—applies to all software development processes. However, we will only consider change impact analysis in a SafeScrum® setting. The challenge is that changes will occur quite frequently in an agile development process and we need a regime that will not slow down this process unnecessarily. SafeScrum® uses the sprints together with the alongside engineering safety activities to uncover and resolve safety issues during development, as close in time to the code creation as possible. See also IEC 61508-1:2010, clause 7.16.

8.2.2 Requirement Changes

When suggesting new requirements or changes to existing requirements, a backlog refinement meeting (also known as requirements meeting) should be held. Relevant experts and decision-makers should be included in this meeting to ensure correct evaluations and quick decisions. Depending on the result, the requirement meeting will result in

- Changes or refinements to the requirements
- No changes
- Changes that require an application condition to be included in, for example, the user manual

8.2.3 Design and Code Changes

In agile development, the need to consider and analyse the potential safety impact of a code change typically originates from backlog refinement meetings where the product backlog is refined based on new, improved knowledge—see Sect. 7.6. This will take place at the start of every sprint. However, it might also happen due to observations in a sprint review meeting.

SafeScrum® uses the alongside engineering safety activities, which run in parallel with the sprints to uncover and resolve safety issues during development as close in time to the code creation as possible. If, however, issues are raised due to significant changes, newly identified hazards, changes in the SRS, or changes in the architecture or software system design—see Sect. 4.1—a more thorough change impact analysis is needed. The cost for the change may also be included. Remember the phrase you learned in school, typically before an evaluation; "If you are unsure, put down your first guess, it has the best chance of being right". Also have in mind Einstein's famous quotation "The intuitive mind is a sacred gift and the rational mind is a faithful servant. We have created a society that honours the servant and has forgotten the gift". This still applies!

The introduction of new code in the sprints calls for re-evaluation of safety and how changes to the code or design comply with the safety requirements. It is important to uncover any problem and to resolve it as soon as possible. Leaving unresolved issues to later stages in development may compromise the whole project since safety is non-negotiable. The change impact analysis decision should be done in a three-step process as follows:

1. The person who will implement the change should consider whether the change is safe—that is, does not affect the safety of the system. If this does not help, go to the next step.
2. If the person who will implement the change does not feel sure about the decision, it should be discussed with the rest of the team.

3. If neither the developer nor the team can decide, the decision should be left to the alongside engineering team.

In order to evaluate whether and how code changes affect safety, the following artefacts or documents may be used:

- The agile hazard log—are we affecting the mitigation of one or more hazards?
- The agile safety case—are we affecting one or more assumptions, arguments or evidences?

Important inputs to the change impact analysis are safety requirements, and related safety stories, safety function descriptions, the system design and trace information.

Figure 8.2 gives a complete overview of all processes related to software changes in a safety-critical system. The loop on the right-hand side—refinement, update stories or improved understanding—is the one that happens most often and is simple to perform. The loop on the left-hand side is related to changes in the requirements. It starts with a change impact analysis which generates the change impact analysis report (CIAR).

After the CIAR, we should consider the agile contract. A good example of such a contract is developed by the Norwegian government—"Agile Software Development Agreement" [4]. This contract could be used as a starting point, also for agile projects that do not involve the government.

The recommendation of this report will be (1) update the agile contract and the SRS or (2) generate a change request (CR), update the contract and then update the SRS. In both cases, the next step is to develop the SRS entries into new user stories and safety stories (Fig. 8.2).

8.2.4 Minor Safety Issues

In order to uncover and resolve minor safety concerns as early as possible, the RAMS engineer is closely involved in the SafeScrum® process. However, he or she is only allowed to resolve minor issues (e.g. issues not resulting in a change of the SRS) to the software being developed. Within a sprint, there are two points in time where the RAMS engineer may assist the sprint team in assessing safety. Firstly, in the sprint planning meeting stories are selected and added to the sprint backlog. Since there may be one or more common actions implemented, we need to do the part of detailed design that will affect more than one user story—see also Sect. 4.1. The RAMS engineer should be present to review suggested design ideas and to assist the team. Secondly, in the sprint review meeting, when resolved stories are demonstrated and reviewed, the RAMS engineer should participate. His or her role is to check that what was implemented in the last sprint is OK with regard to system safety and to safety requirements implemented in the last sprint. This will be a valuable support to the product owner, who is responsible of approving stories as done.

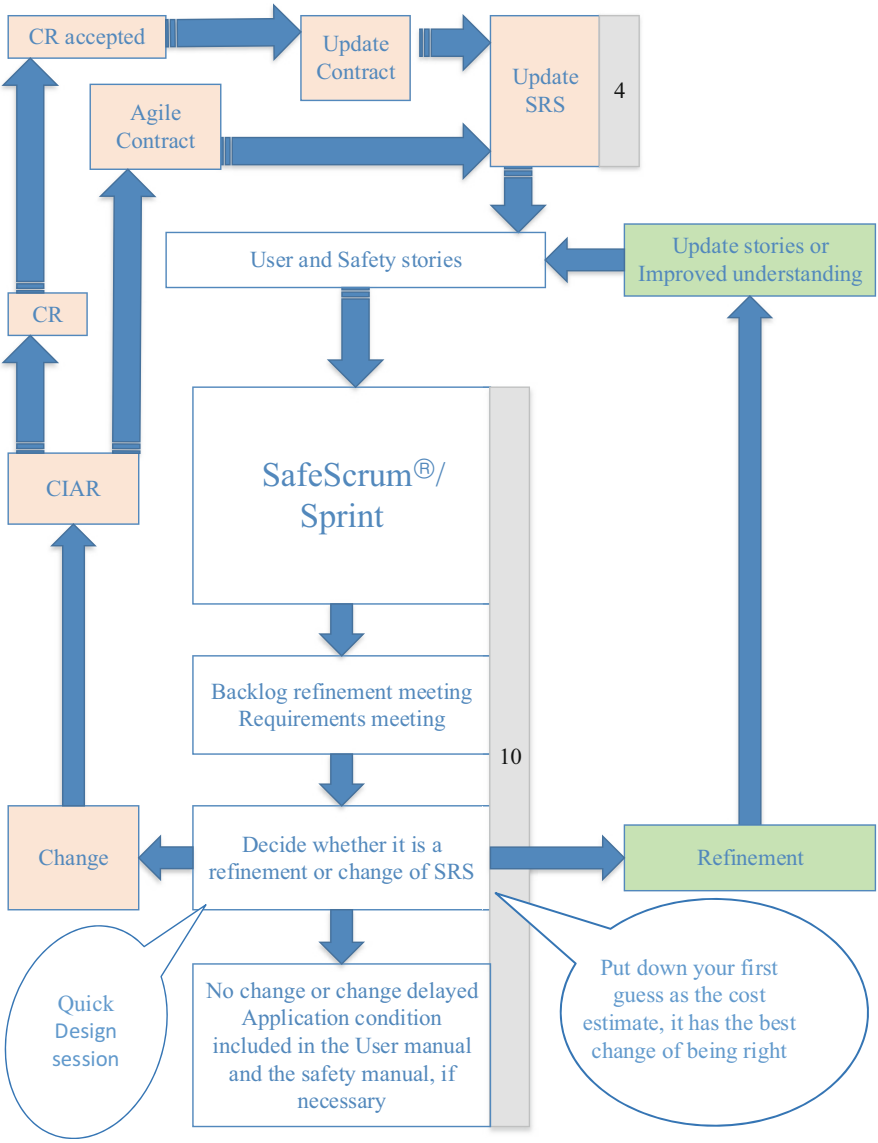

Fig. 8.2 CIA, contract, backlog refinement CR and SRS

8.2.5 Major Safety Issues

In cases of significant changes, where the RAMS engineer is not able to resolve safety issues or where there is a need for a more thorough analysis, the RAMS engineer needs to consult other resources in the organization, for example, other

safety experts. It may also be necessary to perform separate analysis to evaluate the safety impact of, for example, design ideas. This needs to be initiated as soon as possible to provide the product owner and the team with necessary feedback and directions to avoid potential halts in the flow of development. If a major change impact analysis is found to be necessary, the team should—if possible—select stories that are unrelated to the identified issue in the next sprint, pending feedback from the analysis.

8.3 Testing

8.3.1 Classes of Tests

When it comes to testing, we will make a strong distinction between unit testing, which is the developer's responsibility and a part of the sprint workflow (see Sect. 7.2), and integration/module/safety-testing, which are the responsibility of other roles.

8.3.2 Unit Testing

This chapter will focus on unit testing. We will focus on test-first development (TFD)—mainly because of its popularity as it enables good code design and code documentation. The main reason for this is that the tests and the code will be two semi-independent interpretations of the requirements and thus increase the confidence in the resulting code. In addition, it will force the developer to consider in detail what the code should do. Problems with understanding what the code should do should lead to requirement changes and thus increase the quality of the requirements.

TFD is a development practice that embraces the principle of never adding or changing code without first having added or changed the runnable test case that verifies the code's success criteria. Through studies, TFD has been shown to increase the code quality at the possible expense of productivity due to the extended cost to maintain the tests [5]. We believe this focus on quality could present a benefit in using TFD for safety-critical software development, and that the increased trust in the code will benefit the assessment.

Testing during development—TFD or any other approach—will need to use some temporary code to get data into and out of each software unit—stubs, mocks or fakes. A stub is the simplest possible implementation of an interface; a fake is a simple implementation of an interface, while a mock is a more sophisticated version of a fake—it may, for example, return values, perform parameter checks or do some simple computation.

The test of the code—usually a unit-test—is defined before the code itself is developed. By constantly focusing on building tests prior to code, we will gradually

grow up-to-date tests that cover the complete system. One of the benefits of using TFD is that the software is written in smaller units that are less complex and thus more testable, because more consideration is given to design issues [8]. The tests will also include testing the code for error detection, recovery and graceful degradation. The most practical way to test such mechanisms is by fault injection. It also enables simpler regression testing, and acts as an up-to-date documentation of the code. Automated tests can also be used in earlier stages, and cover integration and acceptance tests through tools like Cucumber and FitNesse [6], which can supplement evaluation done in sprint review meetings.

The use of test-driven development will also fit quite well together with safety analysis using Input-Focused FMEA (IF-FMEA) for safety analysis—see Annex B.7. We can start using the IF-FMEA as soon as we have selected a user story and decided which components we will develop. The inputs and outputs are identified based on the relevant stubs, fakes or mocks used in TFD. The IF-FMEA table can then be used both for safety analysis of the added code and to identify new test cases to check any new required barriers. This approach will help us consider safety right from the first sprint and throughout the whole development process. This will in turn help us to create more safe software since the safety concerns will be a natural part of development and an important issue for each sprint retrospective.

When we do test-first we make a set of tests based on the requirements (user stories) currently in the sprint backlog and develop software with the goal that the piece of code currently developed shall pass the tests. However, we will also have several tests developed for previous requirements. In addition, the tests developed for a user story will, in most cases depend on a set of stubs, fakes or mocks. These tests can thus not be used later for system testing but are still relevant for future unit tests. We see two practical ways out of this:

- Organize the user stories in such a sequence that we avoid—or at least minimize—the need for stubs, fakes and mocks that are needed to be able to test a unit. This is called “shift left” testing. Shift left simply means shifting integration testing to the left of its usual position in the delivery pipeline [3]. Even though this can be considered as testing the integrated software sub-system, we are really just testing the last added software, thus doing a unit test. See for instance [3] and Sect. 4.2.
- Have two sets of tests—one for the total system and one for each increment. The first will be a system test that is increased for each sprint, while the other one is a set of tests only relevant for the designated sprint. The system test should be maintained and run by the same persons who do the RAMS validation in the current SafeScrum® model, while the other tests could be the responsibility of the development team. The tests that are only relevant for a stand-alone test of a single component can be thrown away or rewritten to be included into the system test.

If we do not use fully automated testing for each sprint, it is important to retest only what was affected by the last sprint. To achieve this we will use two important mechanisms: (1) connecting tests to user stories and (2) using the trace information.

We need traces from user stories to code and from user stories to tests. This will give us information about which tests are related to which code units. We only need to retest components that are changed or receive input (directly or indirectly) from changed components. By having efficient tools for automation, it is possible to enable regression testing of relevant parts of the system, with increased frequency.

The standard separates strongly between testing a code unit and testing a safety function. The following question-and-answer sequence with a European certification organization illustrates this quite well:
"In our notes from our meeting, I see two messages that somehow don't add up: (1) it may be a problem that the one that makes programs also is the one that makes the tests. [You] should have someone external [to] check/review. (2) On testing in general: Some of the tests should be written by a person who is not the developer of the code to be tested".

- *On issue 1: Is it sufficient that some—a few—of the unit tests are reviewed by another person or does it mean that all unit tests should be reviewed?*
- *On issue 2: Is this only relevant for some tests—e.g. system tests or does this go for all tests—unit test, integration test and so on?*

Answers from certification organization:

- *According to IEC 61508:2010, it is relevant that an independent person make tests of the relevant safety functions. It must be not a person from outside of the company (maybe for train standards!—EN 50129).*
- *The automatic tests can be done by the same person, code review and system tests please from an independent person.*

8.3.3 Software Integration Testing

Integration testing is done outside the sprints, for instance, by a dedicated test department or similar. However, the team should apply continuous integration, meaning that the new or changed code should be integrated with the code master on a frequent basis and that the master is built often, for example, on a nightly basis. This will uncover potential integration issues after introduction of errors as shortly as possible.

8.3.4 Software Module Testing

IEC 61508-3:2010, section 7.4.7, defines software module testing as:

> *"Testing that the software module correctly satisfies its test specification is a verification activity (see 7.9). It is the combination of code review and software module testing that provides assurance that a software module satisfies its associated specification, i.e. it is verified".*

Module tests are defined in the software verification and validation plan, reflecting the requirements in the SRS. Since requirements is used as the basis for defining backlog stories (see 6.4.2), module tests should be defined in a form that can be used in a tool to automate module tests. This means that module tests can be executed frequently throughout the SafeScrum® process and will be important feedback in, for example, sprint review meetings (see 6.9) where the results from a sprint are evaluated. The definition of *module* is unclear in the standard and it is up to each case to define what this means. However, a module should be a part of the system under construction that can be tested as a coherent unit. See also Sect. 1.6 for the IEC 61508:2010 definition of a module and Sect. 3.5 for more on module testing.

There are several tools for automated acceptance testing that may fit this purpose. Cucumber is a tool for automating tests expressed in a behaviour-driven style, meaning that the tests express how the module is used and what the expected result should be. FitNesse is a Wiki-based tool for automated customer tests, which also may be used for automating module tests.

8.3.5 Safety Testing

Safety testing is always needed when developing safety-critical software, whether we are using an agile development method or not. We have included a short discussion here to show how it fits in with SafeScrum®.

In SafeScrum®, safety testing should be done outside the sprints by the RAMS engineer. Safety testing is challenging for two reasons: (1) it is often about what the system must do or not do under certain circumstances and (2) it is often related to the system's behaviour with its operating environment. Thus, we need all the software handling sensor input, control and actuator input before we can do any reasonable safety testing. However, we should do a lot of safety testing before we run the software on the real hardware, with the real sensors and actuators and in a real environment. All this can be simulated—we do not need the real thing, although it would be preferable for two reasons: (1) you do not have to write the extra software to simulate the real environment and (2) you can never be sure that the simulation and the real environment always behave in the same way. As we will see in the two examples shown below, the amount of simulated inputs may be considerable. Some cases may be tested using random value generators for input, for example, the requirement that the system should not hang.

We will use a simple example to illustrate the challenges with safety testing and how they can be handled. For this purpose, we will consider the automatic cruise control (ACC) system, as specified by the standard ISO 15622:2010 (Fig. 8.3).

To test the ACC system, we need to add driver command input, vehicle motion sensors and other vehicle motion and distance sensors. In addition, we need to observe that the information to driver and actuators are correct, based on the requirements as specified by the standard. All this can vary—from a car driver seat with instruments and pedals to everything simulated on a PC.

We get a better idea if we increase the level of details. We see from the next diagram that what you need to simulate will depend on where you put your borders—that is, which modules do currently exist and which do not. In the diagram below, only the components ACC module, break control module and engine control module are available for the safety test. In order to do a safety test at this stage, we need to simulate the instrument cluster, the radar output, break switches, break lights and the break actuators and speed sensors (Fig. 8.4).

The following is a small test example—testing some of the display-functions (Table 8.2).

We can approach this test in several ways. In all cases, we need to first set the ACC system into the required state—active—and we must "set speed"—the speed the car shall hold if there are no obstacles. Then we need to simulate a "forward vehicle detected" with speed, lane and distance information, either via a PC or a via a real radar unit. This should create a "vehicle detected" signal from the ACC system and position information—same lane or another lane. If the set speed will reduce the distance below what is allowed, the speed will be reduced to an acceptable level. The system should also display forward vehicle speed and distance (Fig. 8.5).

The notation is as follows: $d_{\max}$ is the maximum detection range on a straight road, d_1 is the minimum measurement distance—detection but no measurement is required below this distance, d_0—no detection required. These distances are defined by the vehicle speed and set time gap as shown by the following example: $d_1 = \tau_{\min}(v_{\text{low}})\, v_{\text{low}}$.

The example given in Table 8.3 shows a more complex test case. Here we check that the ACC will achieve constant clearance by adjusting the speed and that the

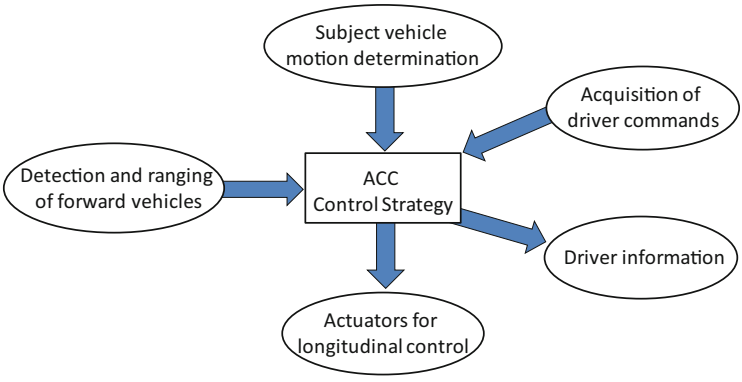

Fig. 8.3 The ACC control system and its environment

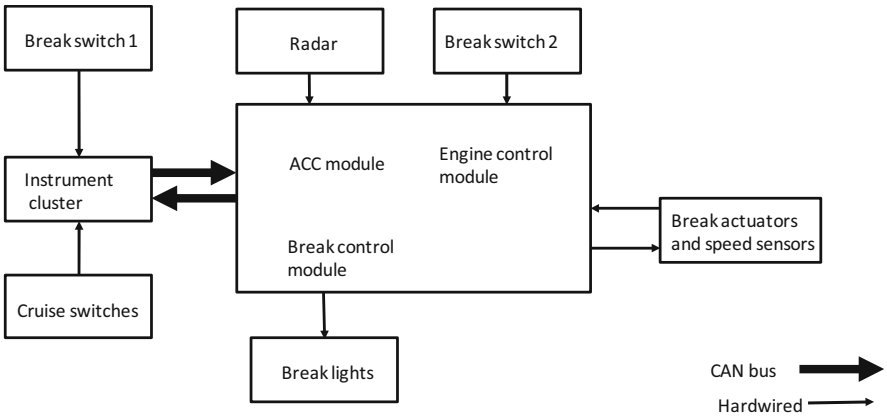

Fig. 8.4 An overview of ACC and its components

Table 8.2 Example safety test

Test ID: 1a		
Test specification: Control Basis		
Reference: Transport information and control system – Adaptive Crouse Controller – Performance requirements and test procedures. ISO 15622:2010		
Input: Sensor		
Expected output: Maintain time gap		
Preconditions: The ACC status shall be active		
Description: When the ACC is active, vehicle speed shall be controlled automatically to either maintain time gap to forward vehicle or the Set speed.		
Action:	Expected result:	Status:
Get a tracking of forward vehicle	Display "vehicle-detected" signal	
Active position of the target vehicle	Display position	
Determine the speed of the target vehicle	Display speed and Clarence	

Fig. 8.5 Important distances for the ACC system

Table 8.3 Safety test example

Test ID: 1b		
Test specification: Control Basis		
Reference: Transport information and control system – Adaptive Crouse Controller – Performance requirements and test procedures. ISO 15622:2010		
Input: Sensor		
Expected output: Maintain set speed		
Preconditions: The ACC status shall be active		
Description: When the ACC is active, vehicle speed shall be controlled automatically to either maintain time gap to forward vehicle or the Set speed.		
Action:	Expected result:	Status:
Get a tracking of forward vehicle	Display "vehicle-detected" signal	
Maintain distance of forward vehicle	Achieve constant clearance	
Determine the speed of the target vehicle	Display speed and clearance	
Achieve the tracking path	Minimize acceleration when it is slope Maximize acceleration when it is on high inclined path	

slope of the road will influence the acceleration. Hopefully, the developers have already realized that the slope indicator is a safety-critical component (Table 8.3).

In the example above, we will also need to involve or simulate the inclination of the road.

8.3.6 Back-to-Back Testing

The idea of back-to-back (B2B) testing is simple and runs as follows: we have several versions of a piece of software, all based on the same set of requirements. We feed that same input to all of them and compare the results. If they all are equal, we will assume that all the pieces of software are correct, otherwise there must be errors in one or more of them. This approach has been suggested in a development concept called N-version programming—see [1]—but is not as hot now as it was some years ago. The B2B testing idea, however, may get more and more popular with increasing use of agile methods with frequent releases.

The SafeScrum® application of the B2B testing is much simpler since we will use it as part of the release strategy. The connection between B2B testing and the SafeScrum® process is shown in Fig. 8.6.

The B2B testing approach is important when we are frequently turning out new versions of the system—probably with every sprint. Most of these new versions are not intended for release—they are just one more error fix or functional extension to

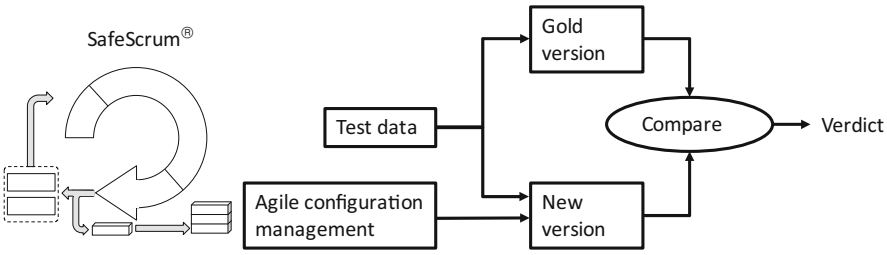

Fig. 8.6 The relationship between back-to-back testing and SafeScrum®

the current system. The next public release might occur in half a year or more. In our case, the gold version is the latest version of the system released to the public.

The test data will concern two sets of requirements: (1) the requirements of the current public version—the gold version—and (2) implementation of the requirements added after the last public release. Case (2) is the most important, since the new version will eventually be the new public version. Handling the tests and the test results are different for these two situations:

1. The test cases for requirement set (1) are the public system's acceptance test cases—the one used for the FAT and the SAT. If these tests also pass with the new version, it means that in these respects, the new version behaves in the same way as the current public one.
2. The test cases for requirements set (2) test the changes to the system. Thus, the tests should fail for the gold version and give correct results for the new version.

The B2B testing approach also allows us to use random input generators. If the results from the gold version and the new version agree, we will assume that it is OK. If the results are different, we need to analyse both versions and decide what is correct and eventually apply the needed corrections. The whole B2B process should be run as follows:

- Use two sets of test cases—one to check adherence to the current public version and one to check the changes.
- Run the tests—the second set should produce different results for the new version.

As much as possible of the above-mentioned process should be automated. Automation can be achieved by using the standard testing tools used for the first acceptance test and then use a difference analysis tool to analyse the differences in the test results.

8.4 Safety Engineering

As shown in Sect. 6.5, the SafeScrum® process, which relates mainly to part 10 of the IEC 61508:2010 safety life cycle model, requires several preparatory activities. These activities are necessary to establish the SRS and to establish important assets to support safety evaluation throughout the SafeScrum® development process.

8.4.1 Safety Analysis

Safety analysis is a fundamentally important concept in SafeScrum®, both as part of preparing a development project (see Sect. 6.5) and during development. There exist a handful of useful techniques and approaches for safety analysis that can be applied. See Annex B in this book for a short overview.

8.4.2 Agile Hazard Log

The agile hazard log (AHL) enables a structured, agile and flexible approach allowing for frequent updates and a shorter time to market. EN 50126-1:1999 has defined hazard log as

> *"The document in which all safety management activities, hazards identified, decisions made and solutions adopted are recorded or referenced*. Our suggestion for a definition of an agile hazard log is *Information on all safety management activities, hazards identified, decisions made and solutions adopted. This should be collected and registered in an adaptive, flexible and effective way"*.

The agile hazard log is constructed based on the initial hazard identification and safety analysis. New hazards will be added to the log as they appear, due to, for example, new or changed requirements, changes to the system's planned operating environment or the discovery of a new hazard, for instance during a daily stand-up or during a sprint review. The agile hazard log serves three important purposes:

- It is an updated repository for all hazards being identified for the current product.
- The associated risk will help us to prioritize the implementation of mitigations.
- When we want to convince the assessor that we have handled all hazards, it is important to be able to (1) refer to the agile hazard log to identify all hazards, and (2) to refer to the mitigations to show that all identified hazards have been dealt with.

Table 8.4 Example lay-out of a hazard log

Identified Hazard	Associated Risk (Consequence)	Existing Mitigation Measures in Place	Current Level of Risk	Further Mitigation Measures	Revised Level of Risk	Action By and When
			Severity Likelihood Tolerability		Severity Likelihood Tolerability	

An agile hazard log shall provide the following information: hazard id, likely consequences and frequencies of the sequence of events associated with each hazard (giving us the risk of each hazard—see Annex B, table 2), and the measures taken to either reduce risks to a tolerable level, or remove the risk for each hazardous event. This covers the first half of the hazard log (Table 8.4).

The second part of the hazard log contains further mitigation measures, the level of risk that this measure will achieve, who will implement it, and when.

Companies introducing agile methods like SafeScrum® should also use an agile hazard log to get the full benefit of an agile approach and at the same time satisfy relevant safety standards such as the EN 5012X series and the IEC 61508:2010 series. The main reasons for introducing the AHL are:

- It is one of the main references in the safety case—see next chapter.
- When introducing SafeScrum®, other parts of the product development process, for example, the hazard log, has to be included to ensure that all the main parts of the development process are agile.
- It will support frequent changes to the system.
- It may facilitate a single source approach for risk management activities.
- It simplifies reuse and transfer of information between stakeholders.

The introduction of the AHL helps to avoid software design errors. Current standards are weak and do not match the current and future heavy focus on software processes. A hazard log that is not adapted to frequent changes may quickly become outdated, in the sense that it no longer represents the true picture of the risks related to the product being developed.

The AHL is developed alongside the product development—that is, in activities performed alongside the sprints. The AHL-related work can be time-boxed together with the sprints, but is normally performed alongside the sprints by the alongside engineering team. The sprint review may include the AHL as a topic when relevant, for example, when new hazards are included in the AHL or when other measures have to be included in the backlog. The development of the AHL should preferably be planned together with other alongside activities like the development of the agile safety case, analysis and independent tests.

The AHL has to satisfy the relevant safety standards. Thus, the requirements for an agile hazard log and an ordinary hazard log are more or less the same. In the EN 5012X series, the main requirements related to hazard identification and hazard processes are included in EN 50126-1:1999 and EN 50129:2003. The requirements

and information in EN 50128:2011 are of little help except that the validator has to *ensure that the related hazard logs and remaining non-conformities are reviewed and that all hazards are closed in an appropriate manner through elimination or risks control/transfer measures.*

The majority of the requirements related to the hazard log in EN 50126-1:1999 and EN 50129:2003 are on the hazard log itself and, to a lesser degree, specific requirements on the process, even though EN 50126-1:1999 states when and in which life cycle phases the hazard log shall be updated or reviewed. According to EN 50126-1:1999, the hazard log shall include or refer to details of:

1. The aim and purpose of the hazard log.
2. Each hazardous event and contributing components, often a limited set of top hazards (typically 5–12 hazards in the railway signalling domain) are defined and a larger number of hazardous events are defined that may lead to a hazard occurring.
3. Likely consequences and frequencies of the sequence of events associated with each hazard. Different approaches exist, for example:
   (a) Detailed calculation by fault tree analysis to determine frequencies (and consequences) for the sequence of events (causes) associated with each hazard.
   (b) Engineering judgement of the consequence and frequency of the sequence events associated with each hazard.
4. The risk of each hazard and risk tolerability criteria for the application.
5. The measures taken to reduce the risks to a tolerable level, or remove the risk for each hazardous event. A number of hazardous events may be controlled by instructions in manuals, operational rules, traffic rules, etc. A potential challenge may be to safeguard that future changes within manuals, operational rules, and traffic rules do not negatively affect risks related to the hazards in the hazard log.
6. A process to review risk tolerability, the effectiveness of risk reduction measures, a process for ongoing risk and accident reporting, a process for management of the hazard log, the limits of any analysis carried out and any assumptions made during the analysis.
7. Any confidence limits applying to data used within the analysis, the methods, tools and techniques used, and the personnel and their competencies that are involved in the process.

It may be challenging to determine the limits of analyses and scope of the hazard log when the system is complex and when there are many actors involved. Each actor may have different responsibilities in terms of development of the system, for example, different actors developing different parts of the system and the operational aspects of the system. In certain cases, there may be several hazard logs that need to interact, for example, different actors may each control their own hazard log.

8.4.3 Agile Safety Cases

Safety cases [9]—also called assurance case or safety demonstration—have, for a long time been required for safety-critical systems in important industrial areas such as nuclear, automotive and railways. The RAMS engineer or one of his or her team members is responsible for the development of the safety case.

Safety case is an efficient method for helping the developing company to focus on the simple but important question "How do you know that your system is safe enough?" The idea of a safety case is not to provide a mathematical or statistical proof, but to argue as one would in a court of law—hence the name safety case.

All too often, development companies have left the important task of creating a safety case till the end of the project. The reason for this has often been that "we need to have complete knowledge of the system before we write the safety case". This has turned out to be a costly solution. It is much more efficient to build the safety case by inserting the information as it becomes available during project development, rather than constructing the whole safety case in retrospect late in the project. The easy way to do this is to build the safety case based on a predefined pattern—see Sect. 8.4.4—"Constructing safety cases". When we have predefined patterns, we can fill in information in the right place as it becomes available.

Safety cases are used in increasingly more domain-specific standards—for example, ISO 26262:2010 for automotive—and we expect that safety cases soon also will be required by IEC 61508:2010, which is a generic standard. We have thus started the work to include safety case construction into SafeScrum®. This is part of our general work towards including all or most of IEC 61508:2010 into SafeScrum®. In order to achieve this, we need to include all safety analysis into the agile process. However, in order to start the safety analysis, we need information such as architecture, operating environment and intended functionality. This might be in conflict with agile development's fear of a "big design upfront". We here present some initial thoughts and attempts for a solution to these problems.

IEC 61508:2010 does not mention safety cases, only safety requirements and safety analysis. Thus, we will use the safety case structure suggested by EN 50129:2003 as our starting point as it is well established and practical to use, also for other domains. This standard suggests that the documented safety evidence for the system/sub-system/equipment shall be structured as shown below. The considerations related to SafeScrum® are added in bold italics for each part.

- Part 1—Definition of system or sub-system/equipment. This shall precisely define or reference the system/sub-system/equipment to which the safety case refers, including version numbers and modification status of all requirements, design and application documentation. Note that these may change during the development process.

  This is done outside SafeScrum® but the definitions may need to be updated later by the alongside engineering team, for example, due to a better understanding later in the project. However, both the functional requirements and

the safety requirements will most likely change and the structure of the safety case must support the handling of these changes.

- Part 2—Quality management report. This shall contain the evidence of quality management. This chapter has many similarities with the ISO 9001:2015 quality management requirements. If the manufacturer has a certified ISO 9001:2015 quality system, this should be mentioned, together with the scope of the certificate A reference to the ISO 9001:2015 certificate should be included.

  The QA role and corresponding activities that are described in Sects. 6.3 and 7.2.3 are closely linked to this part of the safety case and will provide the necessary information.
- Part 3—Safety management report. This shall contain the evidence of safety management.

  Safety management is done by the RAMS engineer and the alongside engineering team. The results are documented by the RAMS engineer and referred to in the safety case.
- Part 4—Technical safety report. This shall contain the evidence of functional and technical safety.

  Technical safety report is written by the RAMS engineer and the alongside engineering team. The results are documented by the RAMS engineer and referred to in the safety case.
- Part 5—Related safety cases—references to the safety cases of any sub-systems or equipment on which the main safety case depends. Part 5 shall also demonstrate that all the safety-related application conditions specified in each of the related sub-system/equipment safety cases are either fulfilled in the main safety case, or carried forward into the safety-related application conditions of the main safety case.

  We need to check that related safety cases are correctly integrated. However, this is taken care of by standard requirements for the safety manual—see IEC 61508-2:2010 and -3. No special SafeScrum® activities are needed here.
- Part 6—Conclusion. This shall summarize the evidence presented in the previous parts of the safety case, and argue that the relevant system/sub-system/equipment is adequately safe, subject to compliance with the specified application conditions.

  No special SafeScrum® activities are needed here.

Large volumes of detailed evidence and supporting documentation should not be included in the safety case or its parts, provided precise references is given to such documents and provided that the base concepts used and the approaches taken are clearly specified. We will focus on part 4—technical safety report—which again consists of six parts. The most important parts are: (2) assurance of correct operation, (3) effect of faults, (4) operations with external influences and (5) safety-related application conditions. These parts are briefly described below.

- **2—Assurance of correct operation** under fault-free conditions (i.e. with no faults in existence), in accordance with the specified operational and safety requirements. Some important aspects are considered below.

- System architecture in sufficient depth to convey a clear understanding of the principles and techniques used.
- Definition of man-machine interfaces and system interfaces.
- Fulfilment of system requirements specification—demonstrate how the operational functional requirements specified in the system requirements specification are fulfilled by the design. All relevant evidence shall be included or referenced.
- Fulfilment of safety requirements specification—demonstrate how the safety requirements are fulfilled by the design. All relevant evidence shall be included or referenced.
- Assurance of correct hardware functionality shall describe the system hardware architecture,
- Assurance of correct software functionality.

In an agile approach, these issues are ensured through cooperation with the customer. This approach is also in line with the agile manifesto.

- **3—Effect of faults:** It is necessary to ensure that the system/sub-system/equipment meets its THR (Tolerable Hazard Rate) in the event of single random fault. It is necessary to ensure that SIL 3 and SIL 4 systems remain safe in the event of any kind of single random hardware faults, which is recognized as possible. Faults whose effects have been demonstrated to be negligible may be ignored.
- **4—Operation with external influences:** This section shall demonstrate that when subjected to the external influences defined in the system requirements specification, the system/sub-system/equipment continues to fulfil its specified operational requirements, continues to fulfil its specified safety requirements (including fault conditions).
- **5—Safety-related application conditions**: Demonstrate that all the safety-related application conditions specified in each of the related sub-system safety cases are either fulfilled by the main safety case, or carried forward into the safety-related application conditions of the main safety case.

8.4.4 Constructing Safety Cases

There are several methods that can be used to present a safety case, for example, the goal structured notation (GSN) method and the structured prose method. The GSN-method has several strengths, for example, a large amount of published patterns, which will simplify the work of developing a safety case. However, a large segment of the relevant industries has used just text. In our opinion, structured text will be an important improvement over plain prose and we will thus start there. The use of GSN should come later. We will take Holloway's work as our starting point. His idea is simple and effective; use the text structure to show the relationships between goals, contexts, strategies, claims, evidences and justifications. The

following example is taken from Holloway's paper [7]—key words are in bold. Note the difference between strategy and argument. The strategy describes which type of argument is best suited for the issue at hand—for example, hazards or design (inspection) or code (testing). The argument is about what we consider as evidence, for example, argue that a certain item in the hazard log has been treated in a satisfactory way.

Claim 1: System is acceptably safe
Context 1: Definition of "acceptably safe"

Claim 1.1: All identified hazards have been eliminated or sufficiently mitigated.
Context 1.1-a: Tolerability targets for hazards.
Context 1.1-b: Reference to current version of the hazard log.
Strategy 1.1: Arguments over all items in the hazard log.
Claim 1.1.1: Hazard H1 has been eliminated.
Evidence: Document reference, for example, to the relevant part of the hazard log.
. . .
Claim 1.1.n: Hazard Hn has been satisfactory mitigated.
Evidence: Reference to code analysis and test results.
. . .
Claim 1.2: . . .
. . .

This notation is simple to read and provides the necessary structure without being overburdened with too much text. It is also simple to update, which is important in an agile setting where we might frequently get new or changed requirements. This might again lead to new risks, and the need for new evidences. It is important to keep just the structure information in the safety case and use references for all information—for example, evidences. In this way, we will have a safety case structure that is easy to read and understand.

In SafeScrum®, developing and maintaining the safety case is the responsibility of the alongside engineering team. Both the hazard log and the safety case will change over time, especially in an agile project. We will start with the hazards found in the hazard log when phase 4 in the IEC 61508:2010 life cycle is finished. After this, necessary updates to the hazard log and to the safety case should be part of the agenda for each sprint retrospective. The structure suggested above makes it simple to add, change or remove items in the safety case. While new hazards can be added to the hazard log as soon as they are identified, the claims, context and evidences must be added later. The need for new evidences will often require new activities, which must be inserted into the sprint backlog—for example, new tests or new analyses. Thus, it is important that we keep a list or library of acceptable evidences related to handling different types of hazards. A possible way to do this is to use the format suggested by table 17 in Annex B. The necessary evidence will in this case be related to "Control or barriers" field. The evidences must be agreed with the assessor and can later be reused. We need information on (1) necessary contexts—what do we

need as context for a specific type or category of claim, and (2) strategies—which strategies are acceptable to the assessor, depending on the type of issue?

By creating and maintaining such a list or library, constructing a safety case will be greatly simplified. It is, however, important that this approach does not make the safety case construction an automatic process. The list is not intended to be a replacement for thinking, it is just a support intended to remove the more mundane parts of the process of building a safety case.

Building a safety case will require a certain amount of resources. As mentioned above, several standards require a safety case while some others are on the threshold of requiring it—for example, IEC 61508:2010. For the rest of us, the important question is whether it is worth it. For people using an agile approach, another important question is how difficult is it to include the building of a safety case into an agile process such as SafeScrum®—see for instance [9].

- Is it worth it?—Yes, definitively. It helps us to be sure that the system is safe. In addition it supports the change impact analysis, since it allows us to identify the supporting arguments and evidences related to component and sub-system V&V activities.
- How difficult is it? If we follow the advices given earlier in this chapter it is a straightforward job. It might be a bit challenging the first few times, but afterwards it will be quite easy.

Thus, all projects that develop safety-critical software should build a safety case, if not for the assessor and the certification, then in order to convince oneself that the system really is safe and to get a good overview on how we have assured system safety. The safety case is also a great asset for later maintenance and development.

The first important activity is to build a safety validation plan based on the safety requirements. Already here, several important questions will surface, such as: How do we validate each safety requirement? The safety validation plan is just the high-level plan. We will refine it and add details when we take the user stories and safety stories out of the product backlog and move them into the sprint backlog. In this way, the safety case will be an integrated part of the project and the safety case document will grow incrementally just like the code.

Already during sprint planning, the safety case and the possible need for new claims or evidence will provide the opportunity for a fruitful discussion in the team, which will increase safety awareness and improve the team's safety culture. In addition, we will get an early focus on the certification process and make all participants understand that it is important and needs to be done.

The downside to all this is that a lot of the work the alongside engineering team expend to build a safety case will not directly contribute to the development of running software and only indirectly contribute to the test and verification activities. This goes for such activities as defining arguments, collecting evidence and cross-referencing the safety case with available documents. A lot of the documentation necessary will have to be written anyway due to standard requirements but it will still require some extra paper work and extra activities that do not benefit the customer directly and thus run counter to the agile manifesto's idea of customer focus.

However, given that we use the appropriate tools, a large amount of the needed information can be provided by the tool chain—see also Sect. 10.3.

Reuse of documents and use of document templates, however, will reduce the extra effort needed for building a safety case. Working with the safety case will increase system understanding and will thus lead to a more efficient process.

8.5 Managing Releases

8.5.1 Introductions

Two issues will influence release management—safety and agile development. The safety issue will require extensive testing and, in many cases, certification before a new release. The agility issue is the agile focus on frequent releases. The frequent releases in agile development are needed, at least internally, in order to get the frequent feedback from the customers—for example, via the product owner. Agile development needs this feedback in order to be efficient.

The problems caused by the two issues identified above can be discussed by splitting releases into two parts—internal releases and external releases. Only the last of these two activities will go to the certification body and then to the customers.

Managing software releases is an important part of the overall development process through which software is made available to and obtained by its users. It includes the process of planning, scheduling, managing and controlling development in all phases and for all platforms. Releases follow one of three approaches [2] where the first one concerns internal releases and the last two concern external releases:

- Development releases aimed at developers themselves for testing and analysis. Independent testers might also be involved here.
- Major user releases based on a stabilized development tree (master)—see an explanation of this term after the bullet list.
- Minor releases used to address minor bugs, security issues or critical defects.

A “branch” is an active line of development. The most recent commit on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch. A single Git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the “current” or “checked out” branch), and head points to that branch—from the Git glossary.

Often, the version that will eventually become the next *major* version is called *the development branch*. However, there is often more than one subsequent version of the software under development at a given time. Some revision control systems have specific jargon for the main development branch but a more generic term is “mainline”.

Releases can be feature-based, that is, releasing a new version when a specific feature is finished. Another option is to follow a time-based strategy, where you release the features that are finished at a specific point in time—for example, every 6 months [2].

The introduction of agile software development methods has led to more frequent integration and releases, leading to what is now called continuous integration where software is integrated and tested as soon as it is uploaded to the integration servers. This fits well with an agile approach as it will provide continuous feedback to the team. Going further, continuous delivery automates the delivery process of the software and minimizes manual affairs and requires the creation of an automated deployment pipeline. However, more research is needed here, especially if a certifying body will be involved.

Last but not least—independent of development process, regression testing is an important part of any release process. Regression testing has two purposes—to show that (1) the latest changes did not introduce an error in already existing functionality and (2) to show that the changes did not re-introduce already fixed errors. Both of these cases are taken care of by the previous FAT, given that it is updated with the tests used to validate the previously fixed errors.

8.5.2 Internal Releases

Internal releases are made to be able to run tests—previous FATs and SATs and new tests for the changes. If the release is not a complete system, and only the software is involved, we can use mocks and stubs to make up for the missing parts of the system. If the hardware is also involved, we can use simulators for important parts such as sensors, actuators and operator interventions. This is known as Hardware-in-the-Loop (HIL) testing and will reduce cost and risk since it allows for early and continuous testing. This goes both for the design and engineering phases and for all internal releases. Note that experience has shown that simulating sensors in a HIL test is risky and should be avoided if possible.

In some application areas, it is common to distribute non-finished versions to beta customers but this is not advisable for safety-critical systems. There is no need to involve the assessor except under special circumstances, for example, when our decision will change the system in such a way that later certification may be difficult. Internal releases can be frequent, even several times a day if needed.

8.5.3 External Releases: Deployment

EN 50128:2011 defines software deployment as transferring, installing and activating a deliverable software baseline that has previously been released and assessed.

External releases are meant for the customers and may only be released after proper testing, analysis and certification. External releases shall come with a release note. The release note shall include all restrictions in using the software. Such restrictions may be derived from, for example, non-compliances with standards, or lack of fulfilment of the requirements. The release note shall also provide information on the application conditions, which shall be adhered to. In addition, it shall give information on compatibility among software components and between software and hardware. Before a software release, the software baseline shall be recorded and kept traceable under configuration management control. The assessor needs to agree for the software to be released.

For later testing and maintenance, it shall be possible to reproduce the software release. In addition, a roll-back procedure (i.e. capability to return to the previous release) shall be available when installing a new software release.

8.5.4 Release Challenges

There are two challenges to the way we currently do release management—new equipment and security fixes. New equipment means changes to the software—for example, new software components to handle a new type of sensors or moving from wired to wireless controllers, for example, sensor connections.

The need for security updates poses a special challenge to the usual way of handling releases. The problem is easy to describe but difficult to handle and goes like this: as soon as a security problem is discovered, it might be published on one or more hacker channels and criminal networks, meaning that the security problem will be known to a lot of people. From then and until it is fixed, the system will be at risk. The longer this state of affairs remains, the higher the risk for a security breach and possible safety risk. Thus, a new release with the needed security fix is an urgent matter. The only possible way out of this problem is to speed up the assessment and certification process. In order to get a clear view of this we put the following two questions to two certifying companies:

- What needs to be re-certified: (1) the whole system, (2) the changes or (3) the sub-system affected by the changes—for example, the communication sub-system?
- Would it be possible to make an agreement with the certifier so that we only certify the change process and report the change to the certifying company?

One of the assessors gave the following, clarifying answer: *I personally do not think that a certified change or development process always leads to a certifiable solution. So just reporting changes in conjunction with a certified change/development process would not be sufficient for me as an assessor. I would always also assess the actual performed changes and evaluate if I agree with the impact analysis results.*

The same assessor also added: *A report of the changes including a classification and impact analysis of the changes is always required. Of course, the certifying company may follow the provided argumentation why there is no impact to the overall safety of a system. For sure, a good analysis with sufficient details and good arguments helps during the certification process. Missing information or inconsistencies tend to cause doubts, and will most likely raise questions at the certification company.*

To sum up—changes to the safety-critical part of a system will require a new assessment and certification. Thus, there is an urgent need for new processes, tools and methods to speed up certification, but that is another story.

References

1. Avizienis, A., & Kelly, J. P. J. (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer, 17*(8), 67–80. %@ 0018-9162.
2. Bjerke-Gulstuen, K., Larsen, E. W., Stålhane, T., & Dingsøyr, T. (2015). High level test driven development – Shift left. In C. Lassenius, T. Dingsøyr, & M. Paasivaara (Eds.), *Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25–29, 2015, Proceedings* (pp. 239–247). Cham: Springer International Publishing. %@ 978-3-319-18612-2.
3. Bjerke-Gulstuen, K., Larsen, E. W., Stålhane, T., & Dingsøyr, T. (2015). High level test driven development–Shift left. In *International Conference on Agile Software Development*. Springer.
4. DIFI. (2015). *Agile software development agreement. Agreement governing agile software development. The Norwegian government's standard terms and conditions for IT procurement (SSA-S)* (p. 46). DIFI.
5. George, B., & Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology, 46*(5 SPEC ISS), 337–342.
6. Hanssen, G. K., & Haugset, B. (2009). Automated acceptance testing using fit. In *Proceedings of 42nd Hawaiian International Conference on System Sciences (HICSS'09)* (pp. 1–8). Hawaii, USA: IEEE Computer Society.
7. Holloway, C. M. (2013). *Making the implicit explicit: Towards an assurance case for do-178c*.
8. Müller, M., & Hagner, O. (2002). Experiment about test-first programming. *Software, IEE Proceedings, 149*(5), 131–136.
9. Myklebust, T., & Stålhane, T. (2018). *The agile safety case.* Springer.

Chapter 9
Documentation and Proof-of-Compliance

What This Chapter Is About

- *We discuss issues related to process and necessary documentation.*
- *What is needed by the manufacturer and the assessors, and why.*
- *We discuss the level of trust between the assessor and the manufacturer.*
- *Reuse of information and documentation.*
- *Use of templates.*
- *Which information can be available as part of tools and which documentation should be documented in, for example, named documents.*
- *An overview of relevant proof of compliance documents.*
- *Which documents are developed by the SafeScrum® team and which documents are developed by the alongside engineering team.*

9.1 Introduction

This chapter deals only with issues related to documentation and proof-of-compliance. For the rest of the adaptation to IEC 61508:2010, see Sect. 9.2.

The problem created by the need to develop a large amount of documents and information when developing safety-critical systems is not a challenge just for agile development—it has been identified as a challenge for all development of safety-critical software. A customer-case shows potential for a 40% reduction in engineering hours on paperwork in a sub-sea development project [1]. In some cases, up to 50% of all project resources has been spent on activities related to the development, maintenance and administration of documents [12]. Thus, a way to reduce the amount of needed documentation effort will benefit all companies that develop safety-critical systems. We are, however, motivated by the focus on simplicity and pragmatism in agile methods and believe that adapting principles from agile software development to the development of safety-critical systems will help to simplify

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_9

the work with the documentation and thus to reduce costs. The three most important ideas are to (1) make use of the short work iterations (sprints), (2) update information, not necessarily documents, frequently, in coordination with development and (3) make as many documents as possible reusable by using a generic format.

In our opinion, the relevant standards overdo their focus on documents, mostly because they overdo their focus on process documentation. It is our experience that a large part of this documentation will only be used for proof of compliance (PoC) which is needed in two cases—for certification and in case the product will be drawn into a court case. Using an agile approach will reduce the amount of in-process documents needed. The reason for this is the improved communication provided by the agile approach. Below, there are two examples. For a more thorough description, see Annex A—Necessary Documentation.

- There is less need for problem reports and documented decisions during development—most of the problems emerging during development are taken care of during the daily standups and sprint retrospectives. Improved communication results in less need of documentation.
- There is less need to document and collect data on process problems—the majority of such problems are taken care of during the sprint retrospectives.

Another factor that will reduce lead-time and cost is to tap into the large potential for reuse of whole or parts of important documents, whether we are using an agile approach or not. This can, however, only be achieved if they are written with reuse in mind.

9.2 Trust

Trust is not specifically addressed by certification and conformity assessment standards that the certification bodies normally have to comply to. However, during assessment work since 1987, we have observed that the level of trust that the assessor has in the manufacturer may affect the level of documentation needed for the certification of the product or system. Agile development emphasizes communication, especially face-to-face communication. This will normally increase the trust between the involved parties. In the standard series evaluated, only ISO/IEC 17021:2011 mentions the level of trust the assessor has in the manufacturer. It states that “Familiarity (or trust) threats: threats that arise from a person or body being too familiar with or trusting of another person instead of seeking audit evidence”.

Both parties should be aware of this threat to ensure that trust is not misused. The ISO/IEC 17021:2011 standard is the only one that mentions the requirements for trust related to the assessor (third party). The level of trust that the assessor has in the manufacturer is a subjective issue, and takes time to achieve, so it is important to discuss the level of details, communication possibilities (which types and frequency of, e.g., physical meetings), possible excessive bureaucracy and pragmatism with the assessor at the beginning of the certification process. The important issue is that the

manufacturer has the information they need to do their job and the assessor to do his job.

Trust as a topic in this respect is closely linked to the level of competence and experience of the personnel. In practice, trust is mainly related to people, not organizations. This has been experienced by manufacturers; when the certification body changed their assessors, it resulted in decreased trust. When this is in place, we can start to build trust based on demonstration of competence and strict adherence to all agreements. Communication between the assessor and manufacturer is of crucial importance.

9.3 Requirements Related to Documentation

9.3.1 Reuse and the use of Templates

In order to reduce the necessary documentation, while, at the same time, remaining able to provide necessary information, we believe that proper adoption of agile software development principles from the Scrum methodology and use of tool support may reduce the costs of documentation. We expect to see two cost-saving effects: (1) it will reduce lead-time and increase the development process flexibility, thus reducing development costs, and (2) it will reduce the number of new documents. However, we do not yet have enough data to show that this will be the case.

When doing modification of an already certified product, only a few documents are new, for example, test reports. Furthermore, these documents can be based on templates or reuse (see IEEE 1517:2010 for more information related to reuse, [4]) or documents may be automatically generated. For reuse, we should use already available templates that have been published in industry papers, for example, [7] or published by organizations developing guidelines such as Misra (www.misra.org.uk) and AAMI (www.aami.org). Some standards, such as ISO/IEC/IEEE 29119-3:2013, include procedures and templates for reports such as test status report, test data readiness report, test environment readiness report, test incident report, minimum test status report and test completion report. Exida has issued a book [5] that includes a template for the safety manual as required by IEC 61508. The topics for a safety manual are presented in IEC 61508-2:2010 (Annex D) and IEC 61508-3:2010 (Annex D).

The challenge with this solution is to keep the process and available documentation in line with the relevant standards' requirements while at the same time gaining the benefits from an agile development process. As described below, we can achieve this through a systematic walkthrough of the relevant safety standards' requirements and only keep the minimum of documents together with an evaluation of which documents can be merged, or information that is needed to meet the standards' requirements. However, the amount of documentation and its format should be discussed and agreed with the assessor at an early phase of the project.

9.3.2 Method When Evaluating IEC 61508-1:2010 Documentation Requirements

In order to evaluate documentation requirements in IEC 61508:2010 as part of preparing this book, we have used the same method as we used earlier to evaluate compatibility and potential conflicts with Scrum [10, 11]. The process consists of the following two steps (see also Sect. 11.2):

1. Check each relevant part of the standard (part 1, Chap. 5) and for each requirement ask "If we use Scrum, will we still fulfill this requirement?" This check is used to move the requirements into one out of three parts of an issues list: "OK"—no further action requirement, "?"—need to be discussed further and "Not OK"—will require changes to Scrum and, in a long-term perspective, to IEC 61508:2010. In addition to the issues list we will also get a lot of input into how to modify the Scrum process in order to reduce the amount of conflicts.
2. Check all requirements that are in the categories "?" and "Not OK" against a modified Scrum process model—in our case SafeScrum®. This will reduce the number of problematic requirements further. In addition, the accompanying discussions will enable us to identify new ways of tackling some of the problems discovered.

This evaluation of requirements has been done as an expert evaluation, including all relevant roles in the assessment: one assessor, two Scrum experts, one safety expert and one representative for a company that routinely need to have their software products certified.

9.3.3 IEC 61508-1:2010 Walkthrough of Chap. 5 "Documentation"

We have evaluated IEC 61508-1:2010, section 5.2—Requirements on documentation. The documentation requirements in IEC 61508-3:2010 are just a reference to part 1 of the standard. The result from the first iteration of the IEC 61508-1:2010, section 5.2 walkthrough was that out of a total of 11 issues, we found that

- Five issues were "OK".
- One issue was "not OK" (5.2.3 below). As a result, Scrum has to be adapted to handle this issue. The adaptation is included in SafeScrum®.
- Five issues needed further investigation—"?" (5.2.1, 5.2.4, 5.2.5, 5.2.10 and 5.2.11) These issues are handled below.

The second iteration focused on the following five issues (see also Annex A of this book):

- 5.2.1. The documentation shall contain sufficient information:
  - Each phase of the overall, E/E/PES and software safety life cycles completed. *These documents will fall in the class reusable documents.*

- Necessary for effective performance of subsequent phases.

  SafeScrum® is mainly performed as part of phase 10 Realization. Anyway, an agile approach should, where possible, also be used for the other phases to ensure optimization of the work involved.
- Verification activities.

  The verification process should use automatic testing tools—for example, *Cucumber (*http://cukes.info*/) or FitNesse (*http://fitnesse.org*/). This will also enable a considerable amount of pragmatic reuse.*

  One of the challenges for SafeScrum®, compared to traditional Scrum, is traceability, which is needed for verification. In order to handle this problem, we have added an extra activity to handle all traceability in SafeScrum®.

- 5.2.3 The documentation shall contain sufficient information required for the implementation of a functional safety assessment, together with the information and results derived from any functional safety assessment.

  This problem is partly taken care of by the SafeScrum® process but the assessor will need more information, which is not available as part of Scrum. This means that SafeScrum® needs to be complemented by normal functional safety assessment.
- 5.2.4 The information to be documented shall be as stated in the various clauses of this standard unless justified or shall be as specified in the product or application sector international standard relevant to the application.

  We should be pragmatic when fulfilling this clause, since this opens up for a wide range of interpretations for what should be accepted as PoC. The most important thing, however, is to discuss this with the assessor ***before*** *the project starts in order to get an agreement on the information that will be needed.*
- 5.2.5 The availability of documentation shall be sufficient for the duties to be performed in respect of the clauses of this standard.

  In order to make all relevant documents available for the assessor we need first of all to register all relevant information. A simple way to do this is to use a whiteboard and to take snapshots. Theses snapshots, together with the date and a list of participants should be accepted as process documentation. For design documents, consistency has also to be checked. When the relevant documents are registered, there exist several tools for sharing information, for example, www.projectplace.com.
- 5.2.10. The documents or set of information shall be structured to make it possible to search for relevant information—for example, by tagging it. It shall be possible to identify the latest revision (version) of a document or set of information.

  All relevant documents must be stored in a project database and indexed properly.
- 5.2.11. All relevant documents shall be revised, amended, reviewed and approved under the control of an appropriate document control scheme.

  The important question here is when—for example, after each iteration, after some iterations or just when we have finished all development iterations. Using

the methods suggested for section 5.2.5 *it is easy to conform to the two first points—revised and amended—while the last two—reviewed and approved—might be problematic in the sense that it will bureaucratize and delay the Scrum process, thus reducing its effect. Part of this could be performed by the new QA role described in this book (see Sects. 6.3 and 7.2.3). These review aspects are normally included in the contract between the manufacturer and the assessor.*

Two important things can be done:

- *Move much of the necessary documents out of the Scrum iteration loop, and consequently, include this as part of the alongside engineering process.*
- *Get an agreement with the assessor as to which iterations need to be included in 5.2.11 and how this can be performed when using, for example, databases.*

9.3.4 IEC 61508-3:2010 Walkthrough of the Normative Annex A

Although annex A (Guide to the selection of techniques and measures) in IEC 61508-3:2010 is not directly related to documents and PoC, it gives an overview of the needed activities and thus indirectly an overview of the necessary PoC. The 10 tables—A1–A10—contains a total of 70 requirements. In order to simplify a walkthrough of these tables we have decided to *assume SIL 2 development*, remove all issues related to maintenance and only consider the activities that are marked as HR—Highly Recommended (although, in practice, some recommended (R) activities should be performed). This reduces the number of issues to 19. The two tables A3 and A4 are only concerned with pre-development activities. Three tables—A5, A6 and A7—are only concerned with testing and the PoCs can be sufficiently covered by the automatically generated test logs. Table A2 is concerned with design activities. In our opinion, the PoC will in some cases be satisfied by whiteboard snapshots or data extracted from a workflow tool such as Jira plus a list of participants. The format of information should however be clarified with the assessor. High-level design—architecture—is decided before we enter SafeScrum®. Using the whiteboard for detailed design has some pros and cons. Pro: quick, can document the design process, not only the final result. Con—may lack the formality achieved by a document. It may also be more challenging to manage versions or revisions.

The only challenge is Table A9 “SW verification”, which is concerned with static and dynamic analyses. When we check the more detailed tables—B2 “dynamic analysis and testing” and B8 “static analysis”—we see that the PoC for the requirements in B2 are covered by the test logs. The only remaining challenges are in B8, which requires analysis of control- and data-flow. This document will have to be done separately (outside SafeScrum®) but only when the system is finished and ready for certification.

9.4 Classification of the Documentation

The relevant documents for IEC 61508-3:2010 are presented in Table A.3 (similar tables exists for Part 1 and Part 2) "Example of a documentation structure for information related to the software lifecycle" in IEC 61508-1:2010. Copy from Part 1:

"Tables A.1, A.2 and A.3 provide an example documentation structure for structuring the information in order to meet the requirements specified in Clause 5. The tables indicate the safety life cycle phase that is mainly associated with the documents (usually the phase in which they are developed). The names given to the documents in the tables are in accordance with the scheme outlined in A.1. In addition to the documents listed in Tables A.1, A.2 and A.3, there may be supplementary documents giving detailed additional information or information structured for a specific purpose, for example, parts lists, signal lists, cable lists, wiring tables, loop diagrams and list of variables".

There are several levels of documentation in a software project. The documents at each level have different sources and different costs but often the same roles, both in the project itself and when it comes to certification. The approach described below could also be beneficial for waterfall projects but it is more important for agile projects, since we expect more frequent builds and releases.

- **Reusable documents**—Low extra costs. These are documents where large parts are reused as is, while small parts need to be adapted for each project and even for each sprint for some documents. If reuse is the goal right from the start, the changes between projects or iterations will be small. For further information about reuse, see IEEE 1517:2010.
- **Combined—**Identify documents that can be combined into one document.
- **Automatically generated documents**—High initial costs but later low costs. These documents are generated for each new project or iteration by one or more tools. Examples are test results and test logs for the testing tool and requirements documents from the RMsis tool.
- **New documents**—High costs. These documents have to be developed more or less from scratch for each new project.

In the table in Annex A of this book, we have classified the documents that are specified in the standards' Table A.3 regarding software in IEC 61508-1:2010.

The main documents are the reports, specifications and plans. As seen from the overview in Annex A (also of this book), these documents should be the focus when trying to reduce the documentation work. Overview of document types as presented in A.3 in IEC 61508-1:2010

The main documents are the reports, specifications and plans. As seen from the overview in Table 9.1, these documents form the major parts of the documentation and as such should be the focus when trying to reduce the documentation work. An overview of possible document classes is shown in Table 9.2. We have used IEC 61508:2010 as an example.

Table 9.1 Overview of Table A.3 SW documents

Documents No. as listed in Table 1 in the Annex of IEC 61508-1:2010	Comments This table could preferably be read together with Annex A of this book
Ten reports (No. 13, 14, 15, 16, 17, 18, 21, 24, 28 and 30)	ISO/IEC/IEEE 29119:2013 is a series of standards for software testing The ISO/IEC/IEEE 29119-3:2013 software testing standard lays stress on documentation and provides standardized templates to cover the entire software test life cycle. An agile approach is also included. ISO/IEC/IEEE 29119-3:2013 includes procedures and templates for: • Test status report • Test completion report • Test data readiness report • Test environment readiness report • Test incident report
Six specifications (No. 1, 4, 5, 8, 9 and 10 are test specifications)	The standard ISO/IEC/IEEE 29119-3:2013 includes both agile and traditional procedures for specifications and examples regarding test design, test case and test procedure
Four plans (No. 2, 26, 27, 29)	Validation, safety (can be based on, e.g., EN 50126-1:2017 [2] or IEEE 1228:1994 "SW safety plans" [3]), verification and functional safety assessment. For further information, see The Agile Safety Plan by Myklebust et al. [9] or a more specific and updated version in *The Agile Safety Case* book [6].
Four instructions (No. 6, 19, 20 and 22)	Examples are: • Development tools and coding manuals • User, operation and maintenance instructions • Modification procedure For further information, see ISO/IEC/IEEE 26515:2011
Two descriptions (No. 3 and 7)	SW architecture design and SW system design
One list (No. 11)	List source code
One request (No. 23)	Request for SW modification Tools exist for software modifications such as the open source tool Bugzilla (www.bugzilla.org) This information can be combined with document/database 25
One log (No 25)	SW modification
One manual (No 31)	Safety manual for compliant items The requirements for the manual are presented in Annex D of IEC 61508-3:2010

9.5 Discussion

The acceptance of a system that has safety-critical components rests on three pillars—agreements with the assessor, trust in the developers, and competent work. This holds independent of standard and development methods applied. The pillars are, however, not constructed independently. In our experience, an agreement

Table 9.2 Classes of documents

Class	Document number	Comments
Reusable	Sixteen documents: 2, 3, 4, 5, 6, 7, 8, 9, 10, 19, 20, 22, 25, 26, 29 and 30	Reusable documents should be made more generic by the manufacturer. For documents that shall be updated as part of several sprints, reuse and extendable solutions are very important. These documents could, e.g., include tables or a point list that are easily updated. For more information, see IEEE 1517:2010.
Combined	Two documents: 2 and 26 Three documents: 8, 9 and 10 Five documents: 13, 14, 15, 16 and 17 Two documents: 19 and 20 Two documents/databases: 23 and 25	Fourteen documents can be merged to five documents, depending on the project and the company. References are simplified when combining documents. The general parts are often the same. The relation between activities, etc., is more visible. However, this, to some extent, depends on, e.g., the size of the project.
Generated	Nine documents: 1, 11, 12, 14, 15, 16, 17, 18 and 28	Several possibilities exist depending on tools allowed to be used by the company.
New documents	Five documents: 6 (new tools), 21 (SW safety validation), 23 (request: SW modification), 24 (SW modification impact analysis), and 25 (log: SW modification).	*Discussions with the assessor:* As part of the Scrum mindset, it is important to reduce the amount of documentation and it is assumed that the assessor should be involved early in the project. What could be a minimum of documentation should therefore be discussed with the assessor before starting to develop any new document. This is also dependent on the product to be certified and the development project. *Templates and examples:* For some documents, templates and examples have already been developed as part of research, standardization and organizational work. See, e.g., "Change Impact Analysis as required by safety standards" [8], ISO/IEC/IEEE 29119-3:2013, and www.misra.org.uk.

with the assessor must come first. This will enable us to settle important questions such as:

- Which parts of SafeScrum® may pose problems later in the project?
- What is accepted as PoC for each activity?
- Which documents and information are needed, in which form and when?

When this is in place, we can start to build trust based on demonstration of competence and strict adherence to all agreements.

Our conclusion is simple—the requirement that we need to certify a system according to a standard cannot be used as an argument against using an agile development process. The problems that exist are not a consequence of formulations of the standard's requirements but are related to what the individual assessor will accept as PoC for an activity.

We have looked into the documents necessary for approval of the software and grouped them according to the opportunity for reuse, combination of several documents into one, documents generated automatically, and new documents. Only a few of the documents are new when doing recertification. In addition, we suggest that new documents should initially be discussed with the assessor, having trust and the agile philosophy in mind to ensure correct level of documentation.

References

1. DNV-GL. (2016). *DNVGL-RP-O101: Technical documentation for subsea projects*.
2. EN, 50126. (1999). *Railway applications – The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS).*
3. IEEE. (1994). *Std 1228 standard for software safety plans.*
4. IEEE. (2010). *1517 standard for information technology – System and software life cycle processes – Reuse processes* (2nd ed.).
5. Medoff, M., & Faller, R. (2014). *Functional safety: An IEC 61508 SIL 3 compliant development process*. exida.com LLC.
6. Myklebust, T., & Stålhane, T. (2018). *The agile safety case*. Springer.
7. Myklebust, T., Stålhane, T., Hanssen, G., & Haugset, B. 2014. Change impact analysis as required by safety standards, what to do? In: *Probabilistic Safety Assessment & Management Conference (PSAM12),* Honolulu, USA.
8. Myklebust, T., Stålhane, T., Hanssen, G.K., & Haugset, B. (2014). Change impact analysis as required by safety standards, what to do? In *Proceedings of Probabilistic Safety Assessment & Management Conference (PSAM12)*, Honolulu, USA.
9. Myklebust, T., Stålhane, T., & Lyngby, N.. (2016). *The agile safety plan.* PSAM13.
10. Stålhane, T., & Hanssen, G. K. (2008). The application of ISO 9001 to agile software development. In *Proceedings of Product Focused Software Process Improvement (PROFES 2008)* (pp. 371–385). Frascati: Springer.
11. Stålhane, T., Myklebust, T., & Hanssen, G. K. (2012). The application of Scrum IEC 61508 certifiable software. In *Proceedings of ESREL*, Helsinki, Finland.
12. Wien, T., Reichenbach, F., Carlson, F., & Stålhabe, T. (2010). Reducing lifecycle costs of industrial safety products with CESAR. In *Proceedings of Emerging Technologies and Factory Automation (ETFA)*. Bilbao: IEEE.

Chapter 10
Tools

What This Chapter Is About

- *A short introduction regarding tools and tools classification.*
- *The importance of tool chains and special requirements regarding safety-critical software.*
- *Special tools for test and analysis.*
- *Requirements for software tools according to IEC 61508-3:2010.*

10.1 Introduction

In this chapter, we briefly discuss tool classification, before diving into the importance of using tool chains in agile development and the special considerations we need to make when we develop safety-critical software. Next, we discuss the use of process tools and tools for testing and code analysis, before ending the chapter with a discussion on the classification of generic tools.

Note that this chapter is not about how to use a specific tool. In order to learn how to use Jira, Stash/Git or any other useful tool, you have to consult the manuals or the tool providers' home pages.

This chapter is co-authored with Børge Haugset, The Norwegian University of Science and Technology.

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_10

10.2 Tool Classification According to IEC 61508:2010

Different kinds of tools have different effects on the executable code that is produced. IEC 61508:2010 classifies tools according to how they affect the software being built. Part 4, section 3.2.11 of the standard defines a software off-line support tool as

> *"a software tool that supports a phase of the software development lifecycle and that cannot directly influence the safety-related system during its run time".*

Software off-line tools may be divided into the following classes:

- *"T1: generates no outputs which can directly or indirectly contribute to the executable code (including data) of the safety related system" (e.g. a process support tool like Jira).*
- *"T2: supports the test or verification of the design or executable code, where errors in the tool can fail to reveal defects but cannot directly create errors in the executable software" (e.g. a tool for automated tests).*
- *"T3: generates outputs, which can directly or indirectly contribute to the executable code of the safety related system" (e.g. a compiler, a code-generating tool or a linked library).*

Part 3, section 7.4.4.5 further states:

"...An assessment shall be carried out for off-line support tools in classes T2 and T3 to determine the level of reliance placed on the tools, and the potential failure mechanisms of the tools that may affect the executable software. Where such failure mechanisms are identified, appropriate mitigation measures shall be taken".

This means that all tools that are used shall be listed, and then evaluated according to their type, that is, whether this particular type of tool is producing or altering code and needs more careful attention, or if mitigation is not necessary.

You should also argue for why the chosen tools are safe to use given their evaluated type (T1, T2 or T3). A T3 tool needs to be assessed, for instance, using the argument "proven in use" where applicable. This argument means that the software has provided "...sufficient product operational hours, revision history, fault reporting systems, and field failure data to determine if there is evidence of systematic design faults in a product" [1]. Given this, code-producing tools without their own IEC 61508:2010 certification can be used within a safety-critical software development tool chain, as long as you can argue that it has been used for a long time and challenges are weeded out and documented. A new version of a tool that previously has been proven in use can, however, not be considered safe, as critical errors may have been introduced. In a non-safety-setting you can easily upgrade

your tool to a new version. However, for the development of safety-critical systems, tools (especially type T3 tools) need to be evaluated and proven to be safe to use. This means that upgrading tools to new versions proves trickier and more expensive in a safety-critical setting than elsewhere.

10.3 Tool Chains and Agile Development

A tool chain is the set of coupled software development tools that are used to create a software product, and can consist of product- as well as process-tools. Output from one tool in this tool chain is often used as input in another tool. Within development of safety-critical products, the use of tools may require an assessment of the tools themselves. This is because the development team, and the assessor, need to be sure that the tools are working as intended and do not create new or hide existing safety issues. Which tools need assessment depends on to what degree they affect the code directly, and we will describe the difference below.

Even though the agile manifesto describes how one, within the agile practice, values *"individuals and interactions over processes and tools"*,[1] the use of tools and processes has become increasingly important within agile software development. This is particularly true when considering the growing use of tools for automation of tasks such as compiling, building or running tests. While is it possible to use SafeScrum® in a development environment with mostly manual tracking of, for example, requirements, we believe that the real benefits will come when applying tools to take out as much manual and error-prone work as possible. Tools can be a more important success factor within safety-critical software development than regular software development because of the increased need for documentation, traceability and consistency.

10.4 Special Considerations for a Safety-Critical Tool Chain

Tool chains are considered a basic need in agile software development. Growing a tool chain with strong bonds help with automation, and most modern software development is heavily dependent upon such tool chains. Automation removes tedious, error-prone and expensive work, making time for more challenging and productive ways to spend developers' time.

The main difference between safety-critical and non-safety-critical software is the special attention paid to documentation of both the development process and the quality of the software product, meaning that the value of proper tool support is even more important for safety systems. One example of this kind of information is that

[1] www.agilemanifesto.org

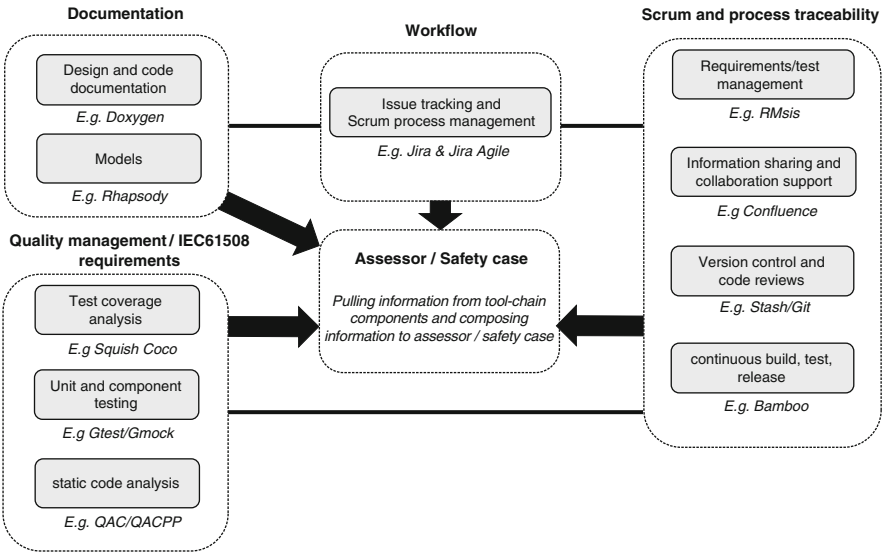

Fig. 10.1 A safety-critical tool chain with Jira-based examples

for SIL 3 and 4, you need a two-way traceability from requirements where they first were described all the way to the resulting code that realizes it.

When we describe a tool chain for supporting safety-critical software development, we will use examples from one of the companies we have cooperated with. They have centred their development on the workflow tool Jira, and their tool chain is shown in Fig. 10.1.

Our intention is not to influence the reader's choice of tools—a tool chain based on, for example, Microsoft's Team Foundation Server (TFS) may work just as well for your needs. Instead, we will focus on the different types of work that needs to be performed in order to successfully release assessed software, and tools that support this. This will then serve as a background for setting up your own tool chain. Even if there are multiple sets of tools shown in the above figure, these can be split into process and product tools.

10.5 Process Tools

10.5.1 Workflow

The workflow tool shall support issue tracking and manage the SafeScrum® process, and can be considered the main development tool hub. It is possible to conduct an

agile process using something else than such a tool, like a whiteboard with yellow stickers, a word processor or more likely a spreadsheet. This, however, will be a costly, manual and error-prone process. In our example, we have chosen Jira and would strongly propose to use this or similar tools.

10.5.2 Scrum and Process Traceability

IEC 61508:2010 necessitates information and traceability of many processes. Below we have listed a few relevant topics:

- **Requirements management:** There needs to be a two-way traceability between all layers of defining and fulfilling requirements, from the SRS onwards to epics, issues, user and safety stories, code and tests. Within the Jira ecosystem, RMsis does just that.
- **System test management:** All requirements need to be connected to system tests. RMsis is a candidate here also.
- **Collaboration/sprint documentation/procedures/how-to's:** There needs to be documentation of who does what—who was at a particular sprint review meeting, how are problems solved, which procedures does the development team follow. Confluence is a wiki software alternative that integrates with Jira and fits the bill.
- **Software version control:** Version control is standard in all kinds of development—also for safety-critical software. If the code is written but not yet reviewed, you can use, for example, Jira combined with Stash/Git to save unfinished work—in this case not-reviewed code—and pick it up later for finalizing.
- **Code reviews:** IEC 61508:2010 may require you to adapt a process where someone else than the author of the code assesses it. A common way of doing so is to let the author of the code submit finished code to a code review, for example, by doing a pull request. Another developer in the team then analyses the code. The result is either an acceptance, or a request for fixes or clarification. Bitbucket is a tool that will support this and enable good traceability of review.
- **Continuous build and test:** Modern software development methods revolve around being able to test and build frequently, while maintaining quality. This is made possible through an extensive set of automated building, testing and deployment tools running on a continuous integration server. One example of such a tool is Bamboo.

10.5.3 Design and Code Documentation

According to IEC 61508:2010, the design of the software as well as the code itself needs to be documented. This needs to be linked to the appropriate parts of the code. One option is to use a tool like Doxygen, a documentation generator that produces documentation from tags and documentation sections in the source code.

10.5.4 UML Models

In IEC 61508:2010, for certain SILs, there are requirements regarding the use of semi-formal methods. UML is one of those languages, and tools like Rhapsody can be integrated into the Jira workflow. There is more on UML in annex C.

10.6 Test and Analysis Tools

IEC 61508:2010 directly describes requirements for the types of analysis that need to be performed (and in some way documented) in order for assessment:

- **Unit testing:** Within agile software development, a unit test tests units of code (such as functions or classes) to ensure quality in the code, that it delivers results that match the specification, and that refactoring does not introduce errors. A unit test is a set of assertions paired with expected results. Tools like Google Test and Google Mock (Gtest/Gmock) let you test your units while mocking (simulating) the rest of the system. Such tests are vital to the software development process. These tests need to be written by the developers themselves.

  IEC 61508:2010 have another definition of unit—see Sect. 1.6—which we have decided to call a functional unit. For a functional unit, IEC 61508:2010 states that tests and code have to be created by independent entities. If someone was embedded deeply enough in the code to be able to write these functional unit tests, they would, however, not be sufficiently independent. Our perception is that functional unit tests still can be made by developers—this should be clarified with the assessor at an early stage. On the other hand, higher-level module and integration tests must be created by someone outside of the development team to enforce independence.
- **Test coverage analysis:** IEC 61508:2010 requires extensive code coverage analysis, such as keeping track of untested code sections, chunks of dead code and test redundancy. One example of such a tool is Squish Coco.
- **Static code analysis:** IEC 61508:2010 may require software code to be analysed (without execution) according to certain criteria. These tools can be considered as performing an automated code review, looking for certain “code smells” like

Table 10.1 Tool types, our suggested classification level and Jira environment examples

Tool type	Classification level	Tool chain example
Scrum workflow management	T1	Jira
UML modeling	T1 (T3 if code is automatically generated)	Rhapsody (Rhapsody has kits for several safety standards including IEC 61508:2010)
Design documentation	T1	Doxygen
Code documentation	T1	Doxygen
Continuous build/test/release server—includes the compiler	T2 / T3	Bamboo
Software version control/code review	T2	Stash/Git
Collaboration/sprint documentation/procedures/ how-to's	T1	Confluence
Requirements/test management	T1 / T2	RMsis, Doors
Test coverage analysis	T2	Squish Coco
Unit and component testing	T2	Gtest/Gmock
Static code analysis	T2	QAC/QA-C++

class size, extensive looping (cyclomatic complexity), duplication of code, etc. Various tools exist for different languages, but one example for C/C++ is QAC/QA-C++. Note that the standard is vague on criteria and that these should be defined in each case. See Sect. 7.7.1 for more details.

10.7 Generic Tools and Their Classification Level

In this chapter, we describe a set of requirements from IEC 61508:2010 that may have implications for your software development. Which ones you need depend on which SIL you aim for. We finish this chapter by summing up the different sets of tool examples that we have identified, along with our perceived classification level. Your classification needs to be in agreement with your chosen assessor (Table 10.1).

Reference

1. Exida. (2018). *Resources*. Available May 2018, from http://exida.com/Resources/Term/Proven-in-use.

Chapter 11
Adapting SafeScrum®

What This Chapter Is About

- *We present a method for adapting SafeScrum® to a development standard.*
- *We show how we have adapted SafeScrum® to three important standards:*
  - *IEC 61508:2010—a general standard used e.g., in the process industry.*
  - *DO 178C:2012—a standard for the development of software for the avionics industry.*
  - *EN 50128:2011—a standard for software in the railway industry.*

11.1 Adapting SafeScrum®

The focus for this book is SafeScrum® and IEC 61508:2010. In this section, however, we will see how SafeScrum® also can be adapted to comply with other standards. This shows that the process defined by SafeScrum® is flexible. As will be seen, all adaptations can be handled by adding features to SafeScrum®. Since SafeScrum® is a minimum safety-critical software development process, there is no need to remove features.

We use the same method here as we have used earlier to evaluate compatibility and potential conflicts with Scrum. (1) Check each relevant part of the standard and move the requirements into one out of three parts of an issues list—"OK", "?" or "Not OK". (2) Check all requirements that are in the categories "?" and "Not OK" against SafeScrum®. This will reduce the number of problematic requirements further.

Sections 9.3.2 – 9.3.4 focus on how SafeScrum® handles documentation and proof-of-compliance. In Sect. 11.2, we will discuss SafeScrum® versus IEC 61508:2010; in Sect. 11.3, we will discuss SafeScrum® versus DO 178C:2012; and in Sect. 11.4, we will discuss SafeScrum® versus EN 50128:2011.

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_11

11.2 SafeScrum® for the Process Domain: IEC 61508:2010

11.2.1 The Adaptation

There has been little experience published on the use of agile development both for use together with IEC 61508:2010 and for safety-critical software in general. A search for relevant literature showed that the majority of hits when using the search terms "agile" or "Scrum" and "IEC 61508" or "safety critical" refer to blogs, courses and discussion fora. Searches with only "Scrum" and "safety critical" gave only 25 hits and a low number of peer-reviewed academic papers. This is as expected since we deal with an emerging topic. All that is published is assessment and analysis of how agile methodology will fit a certification scheme for the development of safety-critical software—little practical experience is published. In addition, most of the work done on IEC 61508:2010 is related to versions published before 2010.

The simple process described in Sect. 11.1 helped us to identify 15 issues, mostly related to documentation and planning. We have four areas of concern and they are addressed as described below (with references to sections in IEC 61508:2010 part 3).

Traceability—1 issue—7.1.2.7. The standard requires traceability from the SRS, to architecture, design, code and tests. This is important, for example, for change impact analysis. Traceability is handled by having two (logical) Scrum backlogs—one for ordinary requirements and one for safety requirements plus a mapping between these two backlogs. In this way, we will see which function is used to implement which safety requirement. In addition, we have a separate activity in each development iteration that is used to develop and maintain traces—see Fig. 8.1.

Design—4 issues—7.4.2.2 b7 and b9 (design properties and design assumptions), 7.4.2.13c (documentation of the design) and 7.9.2.11 a (the adequacy of the design considering the SRS). There is a lot of difference between the agile purists' view and what is done in the real world. A case in point is the work done in the early phases of development. In a survey done by S.W. Ambler for the year 2009 with 280 respondents [1], it was found that

- 79% of all agile projects do high-level initial requirements modelling.
- 88% of all agile projects do some sort of initial modelling or have initial models supplied to them.
- 70% of all agile projects do high-level initial architecture modelling.
- 86% of all agile projects do some sort of initial modelling or have initial models supplied to them.

There is no problem doing high-level design at the start of an agile project and most agile projects do requirements and architecture modelling upfront, using the requirements that are already available. It is important to get the architecture right early since changing the architecture later in the development process will be rather costly. An architectural design is also needed for the initial safety analyses, which are needed to define the safety requirements. In addition, there is nothing in an agile

methodology that will prevent us from starting an agile, safety-critical project with a solid high-level architecture and a good understanding of the available requirements.

Planning—3 issues—7.3.2.1 and 7.3.2.2 (validation planning) and 7.4.2.13 h (configuration of the software): The plans we needed to consider are the detailed development plan and the verification and validation plans. The detailed development plan will consist of a high-level plan for the Scrum process and a detailed plan for each sprint. We will also need a verification and validation plan adapted to incremental development. Thus, instead of *one* verification and validation plan, we will need a sequence and the plans for the early stages of development will need to include development of, for example, mock-ups and simulations.

Documentation and proof of conformance—7 issues—7.1.2.3 (development phase description), 7.4.2.13 d, e, f, g (proof of conformance for reused software), 7.4.2.14 c (specification of data structures) and 7.4.7.3 (documentation of testing): The main question here is what an assessor will accept as sufficient documentation, that is, as proof of conformance. Will, for example, a printout of a user story be accepted as documentation for a requirement? We have two areas of concern—the documentation of the final system and documentation for proof of conformance. All documentation needs that are not related to code development should be moved outside SafeScrum®.

Documentation should be handled by a separate team (part of the alongside engineering team), which works in close connection with the SafeScrum® team and participates in each sprint review and sprint-planning meeting. From the developers, code with comments or using tools like Javadoc or Doxygen should be accepted as documentation.

For proof of conformance for running a test suite for example, the following information could be inserted into a formal document:

- A snapshot of the whiteboard during test planning: What did we want to achieve?
- A printout of the test cases or test scripts: How will we achieve it?
- A printout of the test log or test results: What have we achieved?

This should be accepted as proof of conformance for a testing session. Since documentation—or the lack thereof—always is a bone of contention when we discuss agile software development, we will discuss documentation and IEC 61508:2010 in some more details below.

In this book, we focus on software development and the documentation needed there. However, for the development of a safety-critical system, a lot of extra information is needed, such as safety plan, and validation and verification plans. This is the responsibility of the alongside engineering team, which works in close connection with the SafeScrum® team and participates in each sprint review and sprint-planning meeting. The obvious choice in the alongside engineering team will be the RAMS responsible.

11.2.2 The SafeScrum® Approach to IEC 61508:2010

Just to recap: Our model has three main parts. The first part consists of the IEC 61508:2010 steps needed for developing the environment description and then the safety life cycle phases 1–4: concept, overall scope definitions, hazard and risk analysis and overall safety requirements. These initial steps result in the initial requirements of the system that is to be developed and is the key input to the second part of the model, which is the Scrum process. The requirements are documented as a *product backlog*. A product backlog contains all functional and safety-related system requirements in the form of user stories and safety stories, prioritized by the customer. We have observed that the safety requirements are quite stable, while the functional requirements can change considerably over time. Development with a high probability of changes to requirements will favour an agile approach.

Usually, each backlog item (user stories and safety stories) also indicates the estimated amount of resources needed to complete the item—for instance the number of developer work hours. These estimates can be developed using simple group-based techniques like "planning poker", which is a popularized version of wideband-Delphi.

All risk and safety analyses on the system level are done outside the SafeScrum® process, including the analysis needed to decide the SIL level. Software is considered during the initial risk analysis and all later analyses—once per iteration. Just as for testing, safety analysis also improves when it is done iteratively and for small increments.

The core of the SafeScrum® process is the repeated *iterations*, which are called sprints in the Scrum terminology. Each iteration is a mini waterfall project or a mini V-model, and consists of planning, development, testing and verification. For the development of safety-critical systems with SIL3 and higher, two-way traceability is required between system/code and backlog items, both functional requirements and safety requirements. The documentation and maintenance of trace information is introduced as a separate activity in each sprint. In order to be performed in an efficient manner, traceability requires the use of a supporting tool. Several process-support tools exist that can manage this type of traceability in addition to many other process support functions.

An iteration starts with the selection of the top prioritized items from the product backlog. In the case of SafeScrum®, items in the functional product backlog may refer to items in the safety product backlog. The staffing of the *development team* and the duration of the sprint (30 days is common), together with the estimates of each item decides which items can be selected for development. The selected items constitute the *sprint backlog*, which ideally should not be changed during the sprint. The development phase of the sprint is based on developers selecting items from the sprint backlog, and producing code to address the items.

A practice in many Scrum projects is *test-driven development*, where the test of the code—usually some kind of unit-test [3]—is defined *before* the code itself is developed. Initially, this test is simple, but as the code grows, the test is extended to

continuously cover the new code. The benefits of test-driven development are many. The developer needs to consider the testing of the code before implementation, which helps in clarifying design issues. It also provides a safety harness that enables regression testing, and provides documentation of the code.

A sprint usually produces an *increment*, which is a piece of the final system, for example, executable code. It could also be something completely unrelated to code, like writing documentation or performing tests. The sprint ends by demonstrating and validating the outcome to assess whether it meets the requirements stated by the items in the sprint backlog. Some items may be found to be completed and can be checked out while others may need further refinement in a later sprint and goes back into the backlog. To make Scrum conform to IEC 61508:2010, we propose that the final validation in each iteration should be done both as a validation of the functional requirements and as a RAMS validation, to address specific safety issues. If appropriate, the RAMS engineer may take part in this validation for each sprint. He or she should also take part in the retrospective after each sprint to help the team to keep safety consideration in focus. If we discover deviation from the relevant standards, the assessor should be involved as quickly as possible for clarifications. Running such an iterative and incremental approach means that the development project can be continuously *re-planned* based on the most recent experience with the growing product. Between the iterations, it is the duty of the product owner to use the most recent experience to re-prioritize the product backlogs.

As the final step, when all the sprints are completed, a final RAMS validation will be done. Given that most of the developed system has been validated incrementally during the sprints, we expect the final RAMS validation to be less extensive than when using other development paradigms. This will also help us to reduce the time and cost needed for certification.

The relevant documents for part 3 are presented in Table A.3 "Example of a documentation structure for information related to the software life cycle" in IEC 61508-1:2010, see Annex A.2—Safety life cycle document structure. How to make a safety life cycle document is described at the start of this annex as follows:

"Tables A.1, A.2 and A.3 provide an example documentation structure for structuring the information in order to meet the requirements specified in Clause 5. The tables indicate the safety life cycle phase that is mainly associated with the documents (usually the phase in which they are developed). The names given to the documents in the tables are in accordance with the scheme outlined in A.1. In addition to the documents listed in Tables A.1, A.2 and A.3, there may be supplementary documents giving detailed additional information or information structured for a specific purpose, for example, parts lists, signal lists, cable lists, wiring tables, loop diagrams and list of variables".

11.3 SafeScrum® for the Avionics Domain: DO 178C:2012

Note that all references to sections in this chapter are related to DO 178C:2012.

The two statements that are most important when discussing DO 178C:2012 and SafeScrum® are found in section 3.2, where the standard states that *"the process of a software life cycle may be iterative…"* and in section 1.4, where the standard states that *"This document recognizes that the guidance herein is not mandatory by law, but represents a consensus of the aviation community. It also recognizes that alternative methods to the methods described herein may be available to the applicant. For those reasons, the use of words such as "shall" and "must" is avoided."*

Thus, the standard is already goal-oriented. As a consequence of this, the standard describes a set of processes and objectives, not a set of activities. This makes it easier to adapt any development process to the standard. The standard states objectives for the planning, process, the requirements process, the software design process, the software coding process, the integration process, the software configuration management process and the software QA process. This is in line with the "no shall or must" attitude—achieve the objectives and we will not tell you how to do it. In addition, it states that *"not every input to a process need be complete before that process can be initiated, if the transition criteria established for the process are satisfied"*. This is clearly in line with agile thinking.

The example shown below—Table A-3—is typical for the tables used to define the processes in DO 178C:2012. Instead of defining how something should be done, it defines the outputs of each activity. Note—it is output, not necessarily documents. Instead of IEC 61508:2010's SIL values, the DO 178C:2012 uses risk grades from A to D with A as the most severe grade. There is also a level E, which means "no requirements" (Table 11.1).

There are two symbols used in the table to define the need for independence for the roles that participate: a filled circle for independent person and an open circle for no independence needed. If no symbol is present, the developers are free to skip this activity. Thus, for item 7—Algorithm is accurate—this activity can be dropped for category D software, can be done by one of the project participants for category C software but has to be done by an independent person—in our case a person outside the SafeScrum® team—for category A and B.

The circle with a number inside—1 or 2—defines the CM control category needed for each output. The contents of these two control categories are as follows:

- CC2—configuration identification, traceability, change control identification, code retrieval, protection against unauthorized use and data retention.
- CC1—in addition to everything in CC2, we need the following: base lines, problem reporting, change control tracking, change reviews, configuration status accounting, media selection, refreshing and duplication and release information.

An assessment by Hanssen et al., [2] revealed that objectives for the software development processes (DO 178C:2012, Table A-2) and testing (DO 178C:2012,

Table 11.1 DO 178C:2012 Table A-3: Verification of outputs of the software requirements process

	Objective		Activity	Applicability by software level				Output		Control category by software level			
	Description	Ref	Ref	A	B	C	D	Data item	Ref	A	B	C	D
1	High-level requirements comply with system requirements.	6.3.1.a	6.3.1	●	●	○	○	Software Verification Results	11.14	②	②	②	②
2	High-level requirements are accurate and consistent.	6.3.1.b	6.3.1	●	●	○	○	Software Verification Results	11.14	②	②	②	②
3	High-level requirements are compatible with target computer.	6.3.1.c	6.3.1	○	○			Software Verification Results	11.14	②	②		
4	High-level requirements are verifiable.	6.3.1.d	6.3.1	○	○	○		Software Verification Results	11.14	②	②	②	
5	High-level requirements conform to standards.	6.3.1.e	6.3.1	○	○	○		Software Verification Results	11.14	②	②	②	
6	High-level requirements are traceable to system requirements.	6.3.1.f	6.3.1	○	○	○	○	Software Verification Results	11.14	②	②	②	②
7	Algorithms are accurate.	6.3.1.g	6.3.1	●	●	○		Software Verification Results	11.14	②	②	②	

Table 11.2 DO 178C:2012 objectives and SafeScrum®

DO 178C:2012 Objective	Agile strategy	Remarks
1. High-Level Requirements (HLRs) are developed	A system is divided into features. Features are divided into stories. Stories consist of HLRs (and their test cases).	Features are client-valued functions. At the end of each sprint, the implemented user stories are used to update the HLRs.
2. Derived HLRs are defined and provided to the system processes, including system safety assessment process	Derived HLRs are not directly traceable to system requirements. They are developed in the same way as HLRs.	Derived HLRs are provided to the system processes to determine if there is any impact on the system safety assessment and system requirements.
3. Software architecture is developed	Start with a high-level architecture and update/refine it at each software release.	Closure activities include a review of the software architecture to make sure it is consistent with the source code.
4. Low-Level Requirements (LLRs) are developed	Develop LLRs by defining conditions and associated actions.	LLRs can be contained in the source code or the unit tests (embedded in the source code).
5. Derived LLRs are defined and provided to the system processes, including system safety assessment process	Derived LLRs are not directly traceable to HLRs. They are developed in the same way as LLRs (see objective 4).	Derived LLRs are provided to the system processes to determine if there is any impact on the system safety assessment and system requirements.
6. Source Code is developed	Develop source code by applying Test-Driven Development (TDD).	Stories are implemented during sprints.
7. Executable Object Code and Parameter Data Item Files, if any, are produced and loaded in the target computer	Develop object code by applying Continuous Integration (CI) and Continuous Delivery (CD).	When a defined set of features is completed, a release will follow.

Table A-6) can be achieved by applying agile techniques. The remaining objectives are either outside the agile process or there are no suitable agile techniques to achieve them. These objectives can be achieved using traditional methods (inspections, reviews, analyses, management records). The table below shows how agile development in general and especially SafeScrum® can handle the DO 178C:2012 objectives (Table 11.2).

In conclusion, agile methods can be used to achieve a subset of the DO 178C:2012 objectives. No prohibitive conflicts have been identified. Annex A of DO 178C:2012 contains 10 summary tables with 71 objectives. The information provided for each objective includes: (a) a brief description, (b) its applicability for each software criticality level, (c) the requirement for independent achievement, and (d) the data items in which the results are collected. Each objective has been assessed to determine how the objective can be met using an agile approach like Scrum and

whether there is a need for extensions beyond what can be considered a plain agile approach [2].

11.4 SafeScrum® for the Railway Domain: EN 50128:2011

11.4.1 Adaptation

Note that all references to sections in this chapter are related to EN 50128:2011.

To check possible challenges when adapting SafeScrum® for EN 50128:2011 compliant software, we have performed a detailed study of parts 5–7 of the standard. Section 8, concerning development of application data or algorithms, has been left out. After having evaluated the standard's requirements in two iterations, just as we did for IEC 61508:2010 in Sect. 9.3.1, we are left with the following sections of the standard, which will be discussed in more details below:

- Section 5:
  - Section 5.1.2—organization
  - Section 5.3.2—life cycle issues
- Section 6:
  - Section 6.1.4—test requirements
  - Section 6.2.4—software verification requirements
  - Section 6.5—software quality assurance
  - Section 6.6—modification and change control
- Section 7:
  - Section 7.1—life cycle and documentation for generic software Section 7.2—software requirements
  - Section 7.4—component design
  - Section 7.5—component implementation and testing

The rest of the requirements in section 5–7 are either outside SafeScrum®—for example, architecture—or already taken care of—for example, safety requirements traceability. Last, but not least, it is important to remind the reader of section 5.3.2.2 in the standard: *"The life cycle model shall take into account the possibilities of iterations in and between phases"*.

11.4.2 The SafeScrum® Approach to EN 50128:2011

Just to make things clear—this is about Scrum (not SafeScrum®) versus EN 50128:2011. However, in many cases, the changes needed to comply with EN 50128:2011 are already present in SafeScrum®.

Section 5.1 is about project organization. We will focus on 5.1.2—Requirements. Subject to assessor's approval, the requirements manager, designer and implementer can be the same person. The main point is that the tester must be independent of the implementer. Since these are all roles and not persons, it is not a problem for agile development—it only requires that the tests should be written by someone else in the team. Note that verifier, validator and integrator are new roles, which must be added to the Scrum team.

Section 5.3 is about life cycle issues and documentation. We will focus on 5.3.2—Life cycle requirements. The standard requires a software quality assurance plan, a software verification plan, a software validation plan and a software configuration plan. The standard requires that these plans shall be maintained throughout the development life cycle (section 5.3.2.4). This activity has to be inserted into SafeScrum® as part of the sprint planning activity.

Section 6.1 is about testing. We will focus on the requirements which are stated in 6.1.4. It is important to note that the standard here deviates from the strict roles defined earlier. Section 6.1.4.1 says *"Test performed by other parties such as the Requirements manger, Designer or Implementer, if fully documented and complying with the following requirements (the rest of section 6.1.4) may be accepted by the Verifier"*. This opens up for developer testing, as is usual in agile development. The important point is to reach an agreement with the verifier before the development starts.

Section 6.1.4.5 gives a detailed set of requirements for a test report. None of these requirements are difficult to implement. The main question is how to do it without slowing down the agile development process unnecessarily. In our opinion, we need two things: (1) a test report template, and (2) an adaptation of a test tool printout—for example, SCRIPT. Requirements 6.1.4.5 d–f can be achieved automatically.

Section 6.2 is about software verification. Section 6.2.4 describes the requirements for this part of the process. This implies a new role in Scrum. Note that section 6.2.4.4 allows a stepwise development of the verification plan. Section 6.2.4.7 requires us to demonstrate that functional performance and safety requirements are met. This fits well with the SafeScrum® separation of concerns. The responsibility described in this section will fit in with the RAMS activity after each sprint.

Section 6.5 (Software Quality Assurance) has been thoroughly discussed in Stålhane [5]. Important issues such as the definition of the life cycle model and entry- and exit-criteria for each activity are a natural part of agile development. The results presented on documentation in agile development of safety critical software by Myklebust et al. [4] are also important here. Configuration management (section 6.5.4.11) is not a part of an agile development process but is still important. This activity has been discussed with our industrial partners and is easy to add to the SafeScrum® process.

Section 6.6 (Change Impact Analysis) is already part of SafeScrum® [6] and can be used as also for EN 50128:2011. The only challenge comes from the requirement in section 6.6.4.2 which requires us to go back to an appropriate phase in case of requirements changes. It is, however, up to the project manager to decide what the

appropriate phase is. Traceability and configuration management information will help in deciding this.

Section 7.1 is about life cycle and documentation for generic software. Section 7.1.2 gives the requirements for the documents needed for generic software development—46 documents in total—which are described in Annex A1. The majority of these documents (40 out of the 46) can, however, be developed outside SafeScrum® even though they will have to be updated after some of the sprints. This is, for instance, true for the six documents listed under the category "Architecture and design". The six documents that need to be considered are all in the categories "Component Implementation and Testing" (documents 18–20) and "Integration" (documents 21–23).

The only document in the category "Component Implementation and Testing" that must be considered during agile development is the software source code verification report (document 20). In our opinion, verification is done as part of RAMS (Reliability, Availability, Maintainability and Safety) and thus outside SafeScrum® unless it is feasible to make it part of the process.

Integration documentation (documents 21–23) must be created stepwise as more and more code is integrated after each sprint. This updating process should either be done at the end of each sprint or be done in separate activities inserted into the product backlog.

Section 7.2 is about software requirements. Section 7.2.1 contains the software requirements objectives. Section 7.2.1.1, if taken literally, would prevent the adoption of agile development since it requires that all safety and system requirements should be ready before any coding is done. However, since the standard already allows iterative development, this requirement should be interpreted as "...before any coding is done in this iteration"—in SafeScrum® meaning "in this sprint". See also requirement in section 5.3.2.2.

Section 7.2.4.2 requires the use of the software quality model defined by ISO/IEC 25010: 2011, which replaces ISO 9126. This is not a problem for development but might be a problem for testing—especially testing maintainability, usability and portability. It is important for any project, agile or not, to agree with the assessor and the customer how these tests shall be performed.

Sections 7.2.4.16 and 7.2.4.17 require that we develop an overall software test specification. In an agile setting, this is done by (1) developing a test specification based on the currently available knowledge, and (2) updating after each sprint where requirements have been changed, added or removed.

Section 7.4 concerns component design. How to relate to this requirement depends on the definition of a component. From the definition given in EN 50128:2011—"*...well-defined interfaces and behaviour with respect to the software architecture...*" it is reasonable to interpret a component as a functional unit—see IEC 61508-4:2010, section 3.2.3: *Functional unit: entity of hardware or software, or both, capable of accomplishing a specified purpose*. Using this definition, component design is part of system architecture and thus done outside SafeScrum®.

Section 7.5 is about component implementation and testing. Section 7.5.4.10 concerns requirements for component verification. As mentioned earlier, verifier is a new role that must be added to the Scrum team.

In order for SafeScrum® to comply with EN 50128:2011, we need the following extensions:

- Appoint a verifier for the project.
- Software quality assurance plan, a software verification plan, a software validation plan and a software configuration plan. The plans shall be maintained throughout the development life cycle. It is thus a part of the sprint planning activity in SafeScrum®.
- Developer testing (e.g. according to test-first development) is accepted if we reach an agreement with the verifier on this before the development starts.
- Demonstrating that functional performance and safety requirements are met. The responsibility should lie with the safety verification activity at the end of each sprint—that is, the RAMS engineer.

We must develop an overall software test specification by (1) developing a test specification based on the currently available knowledge, and (2) update after each sprint where requirements have been changed, added or removed.

Elements of the agile development process are already used or considered for use in several important industrial domains such as automotive, avionics and industrial automation.

When the identified issues are settled, it should be straightforward to use SafeScrum® and still be EN 50128:2011 compliant. It is now important to get one or more companies to try it out in cooperation with the safety assessors to get a reality check of the concepts discussed above. This will allow us to identify possible problems and to make the adjustments necessary for industrial application.

With a little flexibility, there are, in our opinion, no large obstacles for using agile development for safety-critical software in the railway domain. The main challenges are the EN 50128:2011 requirements on organization, detailed planning, documentation and requirements for proof of compliance, which are more strict and detailed than the ones for IEC 61508:2010. Introducing agility here will lead to a process that is better adapted to handle changes that occur in any software development process and give us an incremental process—development, testing and verification—that again will lead to more efficient software development.

The main differences between IEC 61508-3:2010 and EN 50128:2011 are the more stringent requirements in EN 50128:2011 related to organization, that the validator shall give agreement/disagreement for the software release, documentation and competence requirements.

Suggested improvements of EN 50128:2011 are more requirements and information regarding modern software development methods. This is also in accordance with preliminary work performed by the current IEC 61508-3:2010 maintenance committee.

References

1. Ambler, S. (2009). *Agile practices survey results: July 2009*. Ambysoft [online]. Retrieved from http://www.ambysoft.com/surveys/practices2009.html
2. Hanssen, G. K., Wedzinga, G., & Stuip, M. (2017). An assessment of avionics software development practice: Justifications for an agile development process. In H. Baumeister, H. Lichter, & M. Riebisch (Eds.), *Agile Processes in Software Engineering and Extreme Programming: 18th International Conference, XP 2017, Cologne, Germany, May 22–26, 2017, Proceedings* (pp. 217–231). Cham: Springer International Publishing.
3. Koskela, L. (2008). *Test driven*. Greenwich: Manning.
4. Myklebust, T., Stålhane, T., Hanssen, G., Wien, T., & Haugset, B. (2014). Scrum, documentation and the IEC 61508-3: 2010 software standard. In *International conference on Probabilistic Safety Assesment and Management (PSAM)*. Hawaii: PSAM.
5. Stålhane, T., & Hanssen, G. K. (2008). The application of ISO 9001 to agile software development. In *Proceedings of Product Focused Software Process Improvement (PROFES 2008)* (pp. 371–385). Frascati: Springer.
6. Stalhane, T., Hanssen, G. K., Myklebust, T., & Haugset, B. (2014). Agile change impact analysis of safety critical software. In *Proceedings of International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR)*. Firenze, Italy.

Chapter 12
A Summary of Research

This chapter highlights important research in all areas covered by SafeScrum®:

- *Requirements, testing and code refactoring.*
- *Continuous integration, iterative processes and customer involvement.*
- *Planning and traceability.*
- *What will happen in these areas in the near future.*

12.1 Introduction

The purpose of this section is to provide some short insights into some relevant research that has been published on agile development of safety-critical software. This was done by searching relevant sources for *industrial experience* of agile development of safety-critical software that was published in peer-reviewed journals and conferences. Our goal has been to uncover some practical experience that may be taken into consideration when applying agile methods in general and SafeScrum® in particular. Although there are a lot of other, interesting publications, we have chosen to focus on real results from real work done by real people. Addressing a multi-faceted topic, we have searched both relevant software engineering[1]- and

[1]Information and Software Technology, Journal on Systems and Software, Transactions on Software Engineering, IEEE Software, Software: Practice & Experience, Empirical Software Engineering.

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_12

Table 12.1 Topics covered in the literature overview

Topic discussed	Number of papers	Number of references
Requirements, management, stories and backlogs	9	39
Testing	8	40
Code refactoring	7	18
Continuous integration and builds	7	16
Iterative process	7	16
Customer involvement	6	34
Planning	6	19
Traceability	6	19

system safety[2] journals. In addition to journals, we also searched relevant conferences covering agile software engineering[3] and functional safety.[4]

The search was done through relevant indexes and databases where we used the search string "(agile OR xp OR scrum) AND safety" in various formats, depending on the search interface of the index or database.

The search gave a total of 314 hits. After a review of titles and abstracts, a large amount of papers were removed since they obviously showed no trace of empirical research or evidence (removing, e.g. student experiments and conceptual papers). This left us with 97 papers. Out of these, 56 papers contained studies that reported on the combination of agile development and safety-critical systems. Sad to say, based on the work reported in the papers, not many of the identified papers were experience-based or provided empirical information. In a final review, we were left with ten papers on "agility and safety" [1–10].

The analysis was done using the NVivo data analysis tool where all signs of practical experience was marked and coded. This is a ground-up approach, useful to collocate experience and findings across multiple papers and topics. As an example, where a paper reported experience related to requirements management, the text segment was coded as "requirements management". This code was used for the rest of the text. During this process, we created several codes, but have discarded codes covering less than six papers. This coding was done in multiple iterations to ensure a high coding coverage. NVivo was then used to produce an extract of the coded text, one report per code. These extracts were then used as the basis for the following summary. Table 12.1 shows the eight codes that we report on. The columns show the topics, the number of papers containing experience coded with the topic, and the

[2] *Journal of Safety Research, Safety Science, Safety and Reliability, International Journal of Reliability, International Journal of Safety and Security Engineering, International Journal of Reliability and Safety, International Journal of Reliability Quality and Safety Engineering, Journal of System Safety, Open Journal of Safety Science and Technology, Journal of Safety Studies*.

[3] XP, AP/Agile Universe, Agile Development Conference.

[4] ESREL (European Safety and Reliability Conference), SafeComp, ISSC—International System Safety, Conference, Scandinavian conference of system and software safety, RAMS symposium, PSAM (Probabilistic Safety Assessment & Management).

number of text segments that were code with the topics. In the subsections to follow, we will give a short summary and discussion of the issues raised for each of these eight topics.

There are obviously limitations to this rather quick search and our overview which should be taken into consideration: (1) The number of studies with traces of industrial experience is relatively low. (2) As the scope of the research is wide (agile processes applied to development of SCS), reported experience varies. (3) Cases relate to various domains—most from avionics/aerospace. (4) All studies represent single cases. (5) Our interpretations and extracts are subject to bias. (6) New studies may have merged after our search. In the following, we will focus on ten papers on agility and safety that have some empirical foundation. Note that this is not a "how have we solved this in SafeScrum®" chapter. The purpose is to show examples based on extracts from papers of what is going on in the area of agile development for safety-critical systems. We have, however, used the findings as inspiration in our description of SafeScrum®.

In order to present the information we found in an easily digestible way, we have organized the material into areas of interest, according to the areas identified in Table 12.1. Since the papers quoted have different foci, we have organized each chapter as follows:

- All the papers that are quoted in the chapter are referenced at the start of the chapter.
- The contents from the quoted papers that are relevant for the chapter are put together and organized into a coherent piece of text.
- We have refrained from adding our own opinions or viewpoint for each of the topics. In some cases, we have added some text to organize the contribution from each author into a more readable form. Hence, the following text is based on extracts from the referred papers which may be read in total for context and more details.

12.2 Requirements

This section contains contributions from Fitzgerald [1], Paige [5], Rottier [7], VanderLeest [8], Webster [9], and Hanssen [3].

It should not come as a surprise that requirements and problems related to requirements top the list of issues. If the requirements are bad, no process will help. All the papers that fall into the requirements category have one or more of the following foci: prioritizing, tracing, agile requirements management and safety analysis.

We will start with the user stories. Even though the customer should be the main source of user stories, this is often not convenient, especially when we are developing off-the-shelf software. In one case, the company in question used a system engineer and a pilot, thus combining knowledge of the system and development process, and users' needs and expectations. One of the authors stated that stories

need to be at the right level of granularity. Experience has shown that the ability to write user stories will improve over time—it is an experience-related issue. It is important to write the user story acceptance tests before implementation starts and they thus need to be testable.

Another author claims that from the project perspective, the requirements serve more as a project scope than a definition of what is being created (kept at a high level), even though the user stories are defined in collaboration between the customers, the user experience team, and the development team. One case study found that the customer requirements efforts were mostly completed in the initial stages of the project. At least one case study showed that functional requirements might change frequently while safety requirements usually are stable and even reusable between projects and products. In any case, for safety-critical systems, the backlog will be populated from the System Requirement Specification (SRS).

The early requirements (user stories) were used as a basis for early safety analyses, which again gave new, derived requirements. For each requirement, there are three important issues to keep in mind:

- Technical knowledge: "Do we know how to develop this feature?"
- Story volatility: "What is the likelihood and impact of the feature changing?"
- Criticality: "How critical is the feature's role in overall system safety?"

This information will then prompt further interaction with the customers and give some fine-grained rescheduling of planning and development tasks. When it comes to safety requirements, it seems to be a general observation that

- High-level system risks were identified during the initial phase of the project and added as constraints (safety requirement) to the product backlog. Any other risks that became apparent during development were also added to the backlog.
- For each user story, the relevant risks were considered when tasks for the sprint backlog were identified. For these user stories, developers tended to document and mitigate the risks by adding a section to the tests to expose the risk, in the form of a failing test and then ensure that the risk is contained by getting the test to pass.

One paper reported that the company that was their case used Confluence (a wiki for sharing information). They added Confluence macros to generate the necessary reports, which is important in agile development in order to reduce the volume of reports that the developers have to write manually. The generated reports were also used for regulatory and dissemination purposes. However, ongoing effort is required to link the requirements to implementation and design issues as well as to close issues when the implementation effort is complete. In one case, the company used two backlogs, one for functional requirements and one for safety requirements. The relationships between the two backlogs are maintained to keep track of which safety requirements are affected by which functional requirements. When implementing or changing a functional requirement, we know which safety requirements to consider. This is used when detailing requirements—that is, moving requirements from the

product backlog to the sprint backlog, and when requirements are changed based on input from previous sprint reviews.

Although most of the referred papers only discuss user stories, one paper has also introduced safety stories, which

- Is a modified form of user story to capture information related to hazards.
- Have a dual role; constituting product-related evidence, as well as forming a basis for planning future increments.
- Helps documenting the outputs of the safety engineering steps of the process.
- May contain code snippets illustrating how certain failure conditions affect the correct execution of code.
- Provide suggestions on how to mitigate the effects of hazards that were identified, including rationale, etc., to assist in the preparation of the safety case.

One of the authors states that prioritizing is done by the Product Owner and the ScrumMaster together. Another author states that they prioritize safety stories over ordinary user stories, while a third author just states that each user story has its own priority.

Tracing of requirements throughout the development process is important whether we want to comply with safety standards like IEC 61508:2010 or just want to keep control over the requirements implementation. One of the authors suggests the following chain of traces:

Initial requirements → stories → tasks/sub-tasks → design document → source code → code reviews → builds → unit-tests → rework/bug fix → functional/system test → production code.

In one of the companies studied, traceability was ensured by adding all requirements to the defect tracking system. Another company stated that when adding new requirements, tasks, and code, it is important to check that the requirements are linked to issues.

One of the authors reported that it was efficient to use Jira to trace the tasks associated with every user story. Another company used RMsis, a plug-in for Jira, to establish traceability of the requirements management process.

12.3 Testing

This section contains contributions from Fitzgerald [1], Hanssen [3], Paige [5], Rottier [7], VanderLeest [8], and Webster [9].

Testing must be in focus from the start and an early focus on testing is a good investment. Experience from non-agile projects shows that a low focus on testing in the requirements phase leads to ambiguous, un-testable requirements, which again makes development difficult. Some companies have moved towards a test-driven development process by starting with a test-aware development process.

Automated testing will increase the pressure on test suite maintenance. Maintaining a comprehensive test suite allows development to proceed iteratively,

preventing new iterations to compromise what has been achieved in the preceding iterations. This emphasis on testing, as exemplified by XP's test-driven development practice (TDD), constitutes a clear overlap of interests with traditional approaches to building safety-critical systems. A potential conflict between agility and safety culture is the role and rigor of testing. Testing in agile development can be considered overly optimistic, concentrating on tests that confirm expectations, rather than those that will reveal defects. During the development of safety-critical system, there is a far more pessimistic view: IDS 00-56 Issue 3 (UK Ministry of Defense, 2004) explicitly requires a search for "counter-evidence", that is, evidence of faults.

Unit tests should be generated as part of the coding tasks and checked in with the functional code and therefore automatically linked to the code. These tests are then executed during the build/deployment. The build automation is done via a tool, such as Bamboo, which also offers the option to invoke analytical tools, for example, static code analyzers. Code changes and unit tests are run and changes to test results across builds can be easily linked to problematic check-ins of code. Unit tests can for example be done within Jira or similar workflow tools and functional tests are the responsibility of the test team using a specific quality center testing suite. In a typical build, a regression test suite of more than a 1000 unit tests are run, which may take 40–60 min to execute. The regressions test suite is written by the developers over time, and new tests are added for new functionality and defect fixes. Any failures are recorded and emails are sent to the developers and ScrumMaster.

Automated tests and automatic links to code facilitate easy coverage reporting. Tools such as Gtest and Gmoc can be used to manage unit tests. In addition, a quality assurance process may be needed to ensure that the tests are really run. It is important to automate as many tests as possible.

One of the contributing companies did, however, see a problem, namely that software integration with hardware means that most of the software testing had to be done manually. A user story is only considered complete when all tests are completed, meaning that it proved impossible to complete a user story within a sprint—for example, 4 weeks. Initially, the company tried to circumvent this issue by allowing the testing of a user story to happen during the subsequent sprint while the developers worked on something else. This proved, however, to be a bad idea. The problem was solved by investing a lot of effort in automating part of the manual testing using FIT (framework for integrated tests). This reduced the need for manual testing and increased the ability to complete user stories within one sprint.

Some of the positive experiences reported with test automation are that each user story was documented as a Confluence page, including all manual and automated tests required for the user story and the collective test automation made it possible to shorten the iterations from 4 to 2 weeks. For continuous integration, it is important to automate the execution of the test suite on the integrated platform. This test suite should include as many requirements-based tests and system-level tests as can be fully automated and executed within a reasonable time window. Finally, the build runs an automated user interface test that serves as a smoke test for the product to ensure that basic user interface functionality always works. The automated user interface test suite is brittle as the user interface is evolving. Thus, the smoke test exercises only a minimal amount of functionality.

Automatic testing is a "must" for acceptance testing, integration testing and for testing done after refactoring. Conducting all acceptance tests for each iteration is clearly infeasible due to costs. Refactoring may be "safe" by the use of test suites. However, major refactoring may well change the interfaces in the code, thus invalidating (part of) the existing test suite. The containment of refactoring is important when complying with a standard such as DO 178C:2012 because an apparently minor modification to one section of source code could have major impact on requirements documents, design documents; requirements-based tests, or systems tests.

It is important to

- Check the test before integration begins. Once the implementation begins, code is developed and prior to integration into the code repository, a specific checklist for test development must be followed. The checklist includes ensuring unit tests are developed, static analysis defects and build warnings have been resolved. In addition, API documentation must be in place and we have to check whether new or updated versions of external libraries have been introduced, and that a code review has taken place.
- Test the system's performance in the operation environment. It is important to include a performance test suite that monitors memory and CPU usage. For the safety stories, it is important to invest considerable effort in automating the manual tests, for example, using the FIT—framework for integrated tests. The next step of continuous integration is the automated execution of the test suite on the integrated platform. This test suite should include as many requirements-based tests and system-level tests as can be fully automated and executed within a reasonable time window.

In one of the companies, the QA team perform augmented automated tests (tests plus extra information—in this case, for example, which bug is this test related to). One of the purposes of this is to monitor commits in the repository and perform interactive testing and bug verification as changes occur. In addition, they perform interactive testing, which attempts to quickly detect defects after they have been introduced by reviewing the commit, and not only testing the feature and the expected behavior but also areas that could have been impacted by changes. The result from the QA team is a combination of metrics, where dependencies are examined to determine systematic ripple effects and experience with the software and good communication with the developer who committed the changes. The combination of these factors improves the possibility for early detection of more serious problems. This type of testing is an augmentation to automated tests, but does not replace them. In addition to testing by the QA team, there is at least one group testing activity where the entire team spends time exercising the new features and the software in general. This helps to ensure that everyone is aware of the current state of the software and helps to surface additional defects.

A dedicated QA-role produces system test documentation and executes system test scripts in line with required standards and product specification. It is important to document all test results for release review. In one of the reported cases, they have

defined a dedicated QA-role responsible of checking code test coverage using Squish Coco. The QA log should be updated with references to uncovered code. This is checked by the end of each sprint. Uncovered code should be discussed at the sprint review meeting and the team should define corrective actions, like defining tasks to produce tests. According to the standard, the coverage should be high and what some assessors may want is as high as 99%.

In one of the studies, a tool (QUMAS) is used to provide support to customers who adopt a risk-based approach to validation (two steps: assess the risk related to each function failure and allocate the test effort so that the functions with largest risk are validated first) in line with regulatory guidelines, by allowing the customer to leverage the functional testing performed by QUMAS during the agile process. Customer access to this test and associated process information is managed in a controlled manner (the customer may give input to the tests, but the final decision will be made by the developers). The agile development process also links validated builds of the software product with the relevant demonstration package test data. An important conclusion from this company is that tools are needed to establish and maintain traceability of requirements, tests and code. In addition, the assessor requires a link between requirements and tests, for example, by referring to unique requirement IDs in test cases.

One of the papers reports the experience that in order for test-driven development to work efficiently, the test developers must be involved in the development of requirements. This is done to ensure that the requirements being produced are testable at the level necessary. This mitigates the risk of requirements changes due to un-testable requirements being identified during the testing phase of a program.

Test-driven development has earned a certain amount of popularity. In order to establish a test-driven process, testers must be involved in the development of requirements to ensure that the requirements are testable at the necessary level. This mitigates the risk of requirements changes due to un-testable requirements being identified during the testing phase of a program. Continuous integration starts with the automated compiling of the integrated software platform, requirements-based tests, and system-level tests. The continuous building ensures that a change made to one software component does not prevent another component from compiling. The biggest benefit (of an agile process) was considered to be the use of test-driven development along with iterative development of features. This has made the case company to fairly consistently produce high-quality code.

In one case, developers created white box tests including logic used in response to user gestures in the user interface. These tests are run as part of the build, which is triggered by any integration into the repository but can also be run on a development machine to reduce the number of build failures. One of the companies let a build failure close the code repository until the failure is resolved. The build consists of compilation, which will fail not only on compilation errors but also on warnings, checks the public platform APIs to ensure that all APIs are documented and that the set of APIs has not changed, and of course executes all unit tests.

12.4 Code Refactoring

This section contains contributions from Fitzgerald [1], Hanssen [3], Paige [5], and VanderLeest [8].

Refactoring is an important activity in agile development. It is a general opinion among the published papers that continuous integration and systematic refactoring will lead to quality improvement. One of the papers reports that the team's internal QA-role uncovers issues, not only related to requirements, but also to defined quality rules such as metrics, bad code, which also produces refactoring issues.

One of the papers describes how they get refactoring into the process framework by defining refactoring stories. These stories originate from sprint reviews. Refactoring—fixing bad code, or changing code to adhere to rules—are prioritized in the next sprint, when the developers' memory is fresh.

However, frequent changes of requirements lead to frequent changes of parts of the code, which is a potential source of errors if it is not done properly. Refactoring implies rework and may have a high cost due to the need for repeated extensive testing and review. It should thus be reduced in the late stages of development. Automated test suites will be helpful to allow refactoring without large costs. TDD and high test coverage will also enable "safe refactoring", and hence lead to a high-quality design of code (the company does not separate refactoring from other code changes).

Refactoring decisions need to be carefully considered before they are done since an apparently minor modification to one section of source code could have major impact on requirements documents, design documents, requirements-based tests, or systems tests. Thus, we need to adapt the "minimal upfront design" tenet; find the right level of detail, and "good enough" design early.

Refactoring can be risky when we use an agile process in fixed-scope contexts doing simple design with small releases and refactoring. Refactoring involves a degree of unpredictability and traditional agile process countermeasures for managing this risk, such as customer negotiation, are less applicable. In addition, there is a question whether design improvements can actually be carried out safely, without jeopardizing fixed scope constraints.

Refactoring is supposed to be made "safe" by the presence of test suites. However, major refactoring may well change the interfaces in the code, invalidating whole or a part of the existing test suite. In addition, refactoring may invalidate planned worst-case execution time and safety analyses, and prompt further refactoring.

12.5 Continuous Integration and Build

This section contains contributions from Fitzgerald [1], Hanssen [3], Paige [5], Rottier [7], VanderLeest [8], Webster [9], and Wils [10].

Having a good routine for continuous build and integration makes demonstration and frequent customer collaboration easier; one of the studies reported that the agile

development process links validated builds of the software product with the relevant demonstration package test data. Pre-sales personnel can identify features they wish to demonstrate, select the appropriate validated build containing those features and the relevant demonstration package test data to show the new software to potential customers, and be confident that the demonstration will progress smoothly. This was an improvement over previous practice where sales personnel manually had to prepare demonstration material. From the same case we see that unit tests are generated as part of the coding tasks. The unit tests are checked in with the functional code and therefore link to the code automatically. These tests are executed during the continuous build/deployment. The build automation is done via Bamboo, which also offers the option to invoke analytical tools, such as static code analyzers. Code changers and unit tests are run and changes to test results across builds can be easily linked to problematic check-ins of code. Several of the papers mention that the Bamboo-tool (a tool commonly used on non-safety projects) was successfully used for continuous builds, tests and release management.

Continuous integration (every 4 h in one case) ensures that sales and marketing can demonstrate the latest functionality to customers, confident that the software will be fully functional. Furthermore, nightly builds allow users participating in the design process to see the current state of the product and try the new features as well as support developers need to work with the latest code.

Several found that code quality increased as a result of continuous integration. It is, however, a practice that needs to be used properly; one paper makes a note that continuous integration may only be possible if small increments are a realistic proposition.

One paper reports that the use of continuous integration systems has allowed teams to immediately identify any issues where a change to one component impacted another component. This was found to be extremely beneficial in identifying areas of functionally which have been unintentionally left public and have therefore been used by other components.

To balance the view on continuous integration and automated builds, one paper warns that these practices will most likely be unfeasible when the project enters the final certification phase.

12.6 Iterative Process

This section contains contributions from Fitzgerald [1], Hanssen [3], Ge [2], Paige [5], VanderLeest [8], and Webster [9].

A general and quite complete model of an iterative process has been presented by Fitzgerald (named R-Scrum). An important extension from other agile process models is that this model contains the “hardening sprint” as a separate, final sprint that results in a shippable product.

Another presentation of an iterative process has focused on the sprint and the extra activities needed to make the process compatible with IEC 61508:2010—the SafeScrum® process. The extra activities are as follows:

- Safety analysis when taking requirements out of the product backlog or inserting new requirements into the backlog.
- Use an appropriate tool to build trace requirements.
- Communication with assessor and safety manager.

The first part of the process, needed to build the product backlog, is not included in the SafeScrum® model.

One of the papers states that an iteration officially begins with a kickoff meeting where progress on the roadmap is discussed, the previous iteration is demonstrated and we do a roundtable-style retrospective.

Another paper describes a company where an iteration combines three issues: (1) constructing the software, (2) constructing the argument that the software is acceptably safe, and (3) to always have an acceptably safe software system with each release. Note that we need dedicated roles to produce safety arguments—this cannot be done by developers. In some cases, iterations may need to be extended in order to satisfy requirements for producing a safety argument. Without this argument, the software cannot be deployed. In addition to the iterations used to develop the code, they also had a final, dedicated iteration that didn't deliver new code but was used to finish the safety argument for the interactions between modules/packages.

At the end of an iteration, most agile development models require a retrospective and a demonstration. Everyone at the retrospective must provide input on what went well and what can be improved for the iteration process. Using a roundtable approach elicited more response from the team and has resulted in multiple process improvements. The demonstration allows the team to see the current state of the software, as not everyone looks at nightly builds, and also helps put context around the issues being ranked for the next iteration, as there are usually some incremental improvements and bugs that are being ranked.

For software development for EASA (European Aviation Safety Agency) and FAA (Federal Aviation Authority), we have to consider SOI—Stages of Involvements that are the minimum gates where a Certification Authority gets involved in reviewing a system or sub-system. Using an iterative development process, each of the intermediate iterative audits would be much less time-consuming than a traditional SOI due to the smaller scope. The added benefit of more frequent audits is that any issues identified during initial audits could then be mitigated on features that were yet to be implemented, therefore reducing risk and costly rework as the program progresses. This approach also brings the FAA into the program more directly through the full process, which reduces the risks of unforeseen issues arising during the final certification review.

In a way, running the SOI audit on a feature is a dry run of the process—just as a dry run of tests can provide confidence that the formal test run will go smoothly, early intermediate SOIs can provide confidence that the process approach is acceptable to the FAA. This will potentially increase the workload for the DER—the Designated Engineering Representative (DERs are very specialized and are given authorizations to perform approvals of the data (instructions) used to make certain modifications or repairs to aircraft).

Table 12.2 Synchronization of iterations

Iteration	N − 1	N	N + 1
Story engineering and planning	Plan N	Plan N + 1	Plan N + 2
TDD and integration	Develop N − 1	Develop N	Develop N + 1
V & V	Verify N − 2	Verify N − 1	Verify N
Safety analysis	Assess N − 2	Assess N − 1	Assess N
Safety case development	Argue N − 3	Argue N − 2	Argue N − 1
Evaluate and adjust	Feedback from N − 2	Feedback from N − 1	Feedback from N

One of the papers surveyed describes how the six activities, story engineering and planning, TDD and integration, validation and verification, safety analysis, safety case and evaluation, are organized. Table 12.2 below shows how the activities and information for iterations N − 1, N and N + 1 are organized.

During iteration N, the planning consortium, consisting of all stakeholders, prepare and select stories for the next iteration (N + 1) and the next increment is agreed. TDD is conducted on the current increment N, while validating the previous increment N − 1 through acceptance tests run in simulation. Safety analysis and safety case development activities for iteration N were performed on the N − 2 increment; these could in turn lead to derived requirements such as those introduced by safety analysis that could be fed through to the next iteration. At the end of an increment, evaluation and adjusting of the process were performed using feedback and metrics from past iterations; at this point, standard team reviews could also take place.

12.7 Customer Involvement

This section contains contributions from Fitzgerald [1], Hantke [4], Paige [5], VanderLeest [8], Webster [9], and Wils [10].

One of the cases did not have an on-site customer. The surrogate for this role is the Product Owner. The Product Owner and ScrumMaster are deeply involved in sprint planning and sprint review meetings, thus affording an opportunity at 3-weekly intervals for detailed feedback on desirable functionality and how it should be prioritized from the customer perspective. However, the company provides support to customers who adopt a risk-based approach to validation in line with regulatory guidelines, by allowing the customer to leverage the functional testing performed by the supplier during the agile process. Customer access to this test and associated process information are managed in a controlled manner.

The frequent delivery of working software inherent to the agile development process has also has major benefits. Because the software can exhibit functionality which has been prioritized, this can be demonstrated to customers early. For a newly

developed product, several customers purchased the new software in advance of its formal release on the basis of the interim working functionality that could be demonstrated. This would not have been possible under the previous waterfall development process according to the VP Development and Support. Development was found to be more effective through the constant validation of product and sprint backlogs based on feedback from the Product Owner, QA and customers. The frequent releases and active engagement with customers means that customer requests can be facilitated within about 5 weeks. Continuous integration (every 4 h) ensures that sales and marketing can demonstrate the latest functionality to customers, confident that the software will be fully functional.

In another case, the project manager was the organizational interface to the customer in customer projects with respect to commercial topics and features to be implemented, and the product was delivered by the product owner to the external customer after the last sprint of a release.

As a way to ensure stakeholder representation, one paper proposes to put together a stakeholder consortium as a reinterpretation of the traditional customer role—consisting of systems- and software engineers, external bodies, suppliers, etc. In an (experimental) case, two domain experts from industry (a pilot and a system engineer), who developed user stories, carried out acceptance testing, and acted as domain consultants during the iterative process.

In another case (DO 178)—in order to prioritize the many requirements for the project, they used customer (or surrogate) feedback to identify important issues, and then combined the priorities using a weighted customer list. The result is comparable to the agile approach of choosing small iterations (stories) in order of customer importance. The productivity for this lean approach was notable, reducing the number of hours per Source Line of Code (SLOC) from the industry average of 3.4 down to 1.6 h per SLOC. The clients in the case have been involved both on a daily basis (aware of the providers activities and responding to questions) as well as on iteration boundaries to determine the deliverables for the next iteration. This involvement and flexibility has allowed the provider to change the focus of the teams easily to be able to help meet the requirements of the clients.

Another case shows that the customer requirements efforts were mostly completed in the initial stages of the project; however, ongoing effort is required to link the requirements to implementation and design issues as well as close issues when the implementation effort is complete. If there are feature designs that are being delivered, these are presented using a mock-up walk-through. This includes showing the wireframes representing the visual design and describing the new capability, which includes the customer use cases. This improved communication has also extended to the customer as the supplier can easily discuss features and defects with the team in the same room. The customers are still remote; however, audio and video conferencing are frequently used and the suppliers try to reduce the duration of meetings as they noticed the attention span for remote participants is shorter than for people physically present. Thus, they attempt to have shorter remote meetings and save longer design and implementation discussions for in-person meetings or spread them out over several remote meetings.

In another case, the authors propose to keep the requirements changes to a minimum by having the customer write their own acceptance tests. They also state that as the pressure for iterative and customer-driven software development will further increase, the industry will have no choice but to adapt their processes accordingly.

12.8 Planning

This section contains contributions from Fitzgerald [1], Ge [2], Hantke [4], Paige [5], VanderLees [8], and Webster [9].

In one study, the Sprint Retrospective meeting is combined with the Sprint Planning meeting (typically on a Monday) at the start of the sprint and the focus is primarily on improving estimations, using the data from completed tasks in the sprint. The Product Owner and ScrumMaster are deeply involved in sprint planning and sprint review meetings, thus affording an opportunity at 3-weekly intervals for detailed feedback on desirable functionality and how it should be prioritized from the customer perspective. Under the previous waterfall process, sales and marketing were consulted about requirements at the beginning of the project, and the resulting requirements specifications were rigidly adhered to during subsequent development phases. One issue identified by management from the case had to do with the perception of "short termism" in planning-granularity that arises from the agile process. Because the product backlog tends to only include stories that are scheduled in the next two releases, this leads to a feeling that the planning horizon is more short term. Under the previous waterfall process, long-term requirements were identified in the design document to guide development over the longer term. However, the VP Development and Support acknowledged that this long-term view was largely a perception which was not always fulfilled, and the faster cadence of the agile process ensured more flexibility to respond to market changes and more accuracy in planning estimates.

Another case found that the requirement to produce a safety argument structure sometimes will need to override the other requirements of iteration planning in order to ensure that the release of each iteration is acceptably safe. In other words, iterations may need to be extended in duration in order to satisfy requirements for producing a safety argument, since without this argument, the software cannot be deployed.

In another case, the authors explain the agile planning process: Each increment began with the consideration and elaboration of user stories, all of which were elicited in a provisional form during the initial stage of the study. We additionally developed related safety stories, which were primarily associated with the safety analysis stages of development. These also fed into the planning process alongside the user stories. User stories captured the behavioral characteristics of each system feature. Stories for the system (integrated altitude data display system) were developed by liaising with the customers (domain experts), finding out about the technology involved in altitude measurement, and agreeing with the required behavior of the system. Each story included a field called "fitness criteria", which described the associated safety

properties, and other constraints, to which any implementation of the story must adhere. Such constraints are normally elicited as test cases; however, the inclusion of fitness criteria in a story made safety case development activities easier by giving an early indication of the evidence required to support a particular feature. There was no variability in the user stories and requirements; this is fairly typical of safety critical software systems. Planning began with an initial release plan, which was divided into three iterations, each designed to culminate in a working version of the software. A fourth iteration was anticipated (and estimation deferred) for further detection and removal of residual defects. Basic risk management activities were conducted as part of the planning process. For each story, a set of questions were posed in order to assess the severity and likelihood associated with a set of three risk attributes. Unanswered questions were given a high risk value, until variables can be assigned a value by confident answers to the corresponding questions. The number of risk variables used for the stories was deliberately kept low due to scope, and the risk management proposals were not investigated due to limited time. Variables (and questions) assessed included: technical knowledge ("Do we know how to develop this feature?"), story volatility ("What is the likelihood and impact of the feature changing?") and criticality ("How critical is the feature's role in overall system safety?"). Risk management affected development in several ways, prompting further interaction with the customers and some fine-grained rescheduling of planning and development tasks.

In another case, the authors found that fixed-length iterations add consistency to the planning as well as help to prevent "feature creep" because once an iteration's tasks have been set, they should not be changed. The case has been able to incorporate fixed-length iterations as a sub-prime within the traditional DO 178B waterfall development process. The iteration length that is most common on the case projects is 1 week. This is the shortest reasonable iteration length, and is shorter than an ideal iteration length of 2–4 weeks. These iterations allow for consistent planning and scheduling as well as helping to reduce the change in scope during an iteration while providing a clear picture for when it would be possible to incorporate those changes.

Yet another case explains that planning is done at two levels, a high-level plan called a roadmap done as part of the fiscal year funding and planning process to show the expected progress over the next year and more detailed iteration planning, where the exact changes for the iteration are ranked and committed. The roadmap is used during presentations to illustrate what the deliveries will be for the year and is presented with the caveat that the schedule is approximate given the agile process and may change depending on what is done during the stack ranking. This message is sometimes met with skepticism as there are no exact dates for the software; however, the ability to see the progress of the software at least every iteration, has successfully built confidence for both the user stakeholders and management. The roadmap is developed by the project management and customer representatives where project goals are evaluated against funded development resources. This spans several parts of the organization and thus the roadmap must incorporate features for each stakeholder. The roadmap is consulted during iteration planning to help with prioritizing, specifically this helps provide objectivity when new features are suggested. Iteration planning, in contrast to roadmap planning, is done at the beginning of each iteration. The

case company has adopted a 3-week iteration. Software versions are added to the defect tracking system (Jira) to represent each iteration. Prior to an iteration anyone can add issues to any software version bucket that has not yet been ranked. Issues can represent software functionality or defects that need to be fixed, but also track design tasks for the user experience team, automated testing functionality by the quality assurance team, as well as documentation tasks.

Another case summarizes that the agile planning and development process allows the most important features and bug fixes to be prioritized frequently and thus delivered quickly. This helps bolster both customer satisfaction and confidence and allows continued development of the project.

12.9 Traceability

This section contains contributions from Fitzgerald [1], Hanssen [3], Webster [9], and Paige [5].

End-to-end traceability is a significant overhead in regulated environments. Traceability is often accomplished using spreadsheets that are printed and subsequently manually updated. Traceability is arguably the area in which the agile development process has had the most impact. Combining traces and agile development was, as one of the companies' VP of Development and Support characterized it, "living traceability" since there is complete transparency in the development process at any point in time.

The idea of living traceability is also brought up by another author: According to the VP of Quality and CRM, the final QA release process is much more efficient using an agile process, than when following a waterfall process. "QA audits are done at the end of each sprint which allows for improved visibility, traceability and measurement so we have no unexpected exceptions to address at final release. We are just confirming the final release". This mode of "continuous compliance" is greatly facilitated by the traceability afforded by the toolset used in the project.

When it comes to traceability, tools are almost a necessity. It is barely possible to have traceability using a manual approach—but just barely, and not if you go through a lot of code updates. In the past, documents and artefacts were produced periodically and collated to produce traceability evidence. Now it is possible to have full end-to-end traceability established by the toolset. Links are automatically established as developers check in code that implements a certain task. Should a developer check in code without linking it to a task, the automated check will identify this as an error. Initial requirements can be traced to stories, and in turn to tasks and sub-tasks, to design documentation, to source code, to code reviews, to builds, to unit tests, to rework and bug-fixes, to function and system testing, and to production code.

Furthermore, the toolset can be interrogated to trace which build fixed which bugs, and which build implemented which functionality. A tool chain can be used to enable traceability from requirements, to code and to tests.

For one of the companies that were involved in one of the papers, it was important to know the required level of traces. The question to the assessor about traceability of safety-related requirements was as follows: Is it sufficient to have a trace between documents or should it be possible to trace issues down to sections, pages or lines in the text? The assessor's answer was that he requires a link between requirements and tests, for example, by referring to unique requirements ID in test cases. The company's decision was that this level of trace should be handled by a dedicated requirements management tool (RMsis) linking requirements to tests that validate them, as well as linking requirements and tests to design and code.

They have, however, identified a need to verify manually that this is done correctly and to make necessary corrections. The QA role shall continuously verify that traceability is kept up-to-date, and verify that all steps of the process are done.

In another company, all requirements are added to their defect-tracking system so that implementation and design issues can be linked, which ensures traceability. In order to facilitate traceability, issues scheduled for completion during the iteration use the linking system. Customer requirements, which are provided as issues, are linked to design tasks; a design task is linked to one or more implementation issues as well as the design documents. The implementation issue is linked to the source code repository (Subversion) commits that comprise the issue as well as a technical specification. This extensive linking provides an audit trail, which tracks customer requirements through implementation. The change tracking system that is part of Jira, along with the discipline to document the context of the change, meeting notes, provides a mechanism that has been received favorably during a CMMI audit process.

One of the papers describes a company where all projects undergo external audits of their development process about once per month. The extra transparency afforded by the implementation of their agile development approach has engendered further confidence to the extent that audits may now take place without requiring the attendance of the product manager and test manager. Furthermore, audits, which used to take 2 days, are now being completed in less than a day, often with no open issues to respond to, and resounding approval from audit assessors who appreciate the complete transparency and flexibility afforded by the living traceability, allowing them to interrogate aspects of the development process at will. The automated traceability also better supports the impact assessment from the QA side, when applying change to existing verified functionality.

They have also observed that compliance is more immediate and evident in real time—continuous compliance as we have labelled it here. The concept of living traceability has been coined to reflect the end-to-end traceability that has been facilitated by the toolset that has been implemented to support the agile development process.

12.10 The Near Future: DevOps

This chapter contains short descriptions of interesting techniques and methods that are not yet part of SafeScrum® but are still of interest, since they already are used outside our process and will be increasingly important in the future.

To give a quick summary first—It is all about communication!

The term DevOps stems from the combination of two processes—development and on-site operation. However, it is not intended to be a process. The eBook from New Relics calls it a culture or a movement. In [1], they state:

"DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach. DevOps emphasizes people (and culture), and seeks to improve collaboration between operations and development teams. DevOps implementations utilize technology—especially automation tools that can leverage an increasingly programmable and dynamic infrastructure from a life cycle perspective".

DevOps can be considered an extension of agile development. Agile development has as one of its goals to improve communication between developers, testers and customers. DevOps extends the team by also including site operations in the process. This will benefit both developers and operations. Operations will be able to bring their problems to the attention of the developers quicker and thus get the problems solved earlier. Developers will get a better understanding of the operations problems and the consequences of delivering systems containing errors and thus be more aware of such problems and handle them in the development process.

Including site operations into the development process means that hazards that can only occur during operation are also considered, identified, included in the requirements and catered to, for example, by building barriers or implementing mitigation procedures. Thus, the operations hazards concerns are handled like all other hazard concerns. Involving the site operations organization into the development also enables us to get information on problems and near misses that occur. This will help us to make new and safer products in new releases.

References

1. Fitzgerald, B., Stol, K.-J., O'Sullivan, R., & O'Brien, D. (Eds.). (2013). *Scaling agile methods to regulated environments: An industry case study*. San Francisco, CA: IEEE Computer Society.
2. Ge, X., Paige, R. F., & McDermid, J. A. (2010). An iterative approach for development of safety-critical software and safety arguments. In: *Agile Conference (AGILE), 2010*.
3. Hanssen, G. K., Haugset, B., Stålhane, T., Myklebust, T., & Kulbrandstad, I. (2016). Quality assurance in Scrum applied to safety critical software C3 – Lecture notes in business information processing. In H. Sharp & T. Hall (Eds.), *17th International Conference on Agile Processes in Software Engineering and Extreme Programming, XP 2016* (pp. 92–103). Switzerland: Springer.

4. Hantke, D. (2015). In T. Rout, R. V. Oconnor, & A. Dorling (Eds.), *An approach for combining SPICE and SCRUM in software development projects, in software process improvement and capability determination, spice 2015* (pp. 233–238). Berlin: Springer.
5. Paige, R. F., Galloway, A., Charalambous, R., Ge, X., & Brooke, P. J. (2011). High-integrity agile processes for the development of safety critical software. *International Journal of Critical Computer-Based Systems, 2*(2), 181–216.
6. Pelantova, V., & Vitvarova, J. (2015). Safety culture and agile. *MM Science Journal, 2015* (October), 686–690.
7. Rottier, P. A., & Rodrigues, V. (2008). Agile development in a medical device company. In *Agile, 2008. Agile '08. Conference*.
8. VanderLeest, S. H., & Buter, A. (2009). Escape the waterfall: Agile for aerospace. In *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*.
9. Webster, C., Shi, N., & Smith, I. S. (2012). Delivering software into NASA's Mission Control Center using agile development techniques. In *Aerospace Conference, 2012 IEEE*.
10. Wils, A., Van Baelen, S., Holvoet, T., & De Vlaminck, K. (2006). Agility in the avionics software world. In P. Abrahamsson, M. Marchesi, & G. Succi (Eds.), *Extreme programming and agile processes in software engineering, proceedings* (pp. 123–132). Berlin: Springer.

Chapter 13
SafeScrum® in Action: The Real Thing

What This Chapter Is About

- *First we present a company's development process.*
- *Then we present the workflow plus some example screen shots.*
- *As a last part, we present some screen shots from the sprint review meeting.*

13.1 Introduction

This chapter is a short description of how SafeScrum® is used in a company that we have been working with to develop and test some of our ideas. Thus, the reader should be aware that some of the terms used here are different from the terms used in the preceding 12 chapters of this book, for example, the term "task" is replaced by the term "issue". In addition, some of the terms used are related to the tools used by this company—for example, Jira and RMsis.

We will base this chapter on the process model shown in Fig. 13.1. Sections 13.2–13.4 discuss the content of the three main parts as indicated in Fig. 13.1. The process shown in the model is just one example of how the process can be organized and is based on the practice when developing an IEC 61508:2010 SIL 3 fire alarm system.

The sections below are based on a set of screen dumps from a real project using SafeScrum® and Jira. Note that these are only intended as examples of tools and we do not explain the context, details and terminology from the case project they have been used in—that would be too complex. Also, each case needs to put together the best possible combination of tools. Most of the following examples are based on tools in the Atlassian family—but there is a wide variety of tools from other vendors that may be equally relevant. We recommend that a project using SafeScrum® start out with the main parts of the tool chain, such as workflow management and requirements management. Tools should be configured and the tool chain expanded continuously, based on experience and specific needs of each company.

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8_13

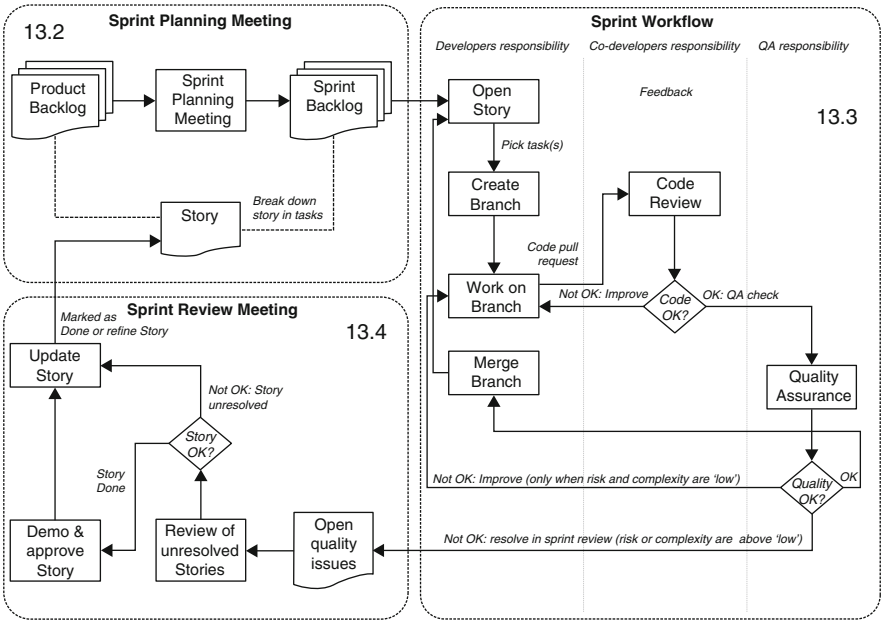

Fig. 13.1 The process map

13.2 Planning the Work

As mentioned earlier, there are two levels of planning—see Figs. 6.1 and 6.4 in Chap. 6. Relating to Fig. 6.4, everything before the SafeScrum® sprint is considered done, although it might have to be updated later. However, these updates are done by the alongside engineering team and is not the responsibility of the developers.

We start with the system's requirements as they are documented in the RMsis tool. The requirements below are related to dual safety—"All loops controlled from either Primary or Secondary (unit)". The RMsis requirements are used as input to the product backlog—see Fig. 13.2.

The requirements from RMsis are reformulated as user stories in Jira as shown in the process depicted Fig. 13.2. An example of a user story for "dual safety" is shown in Fig. 13.3.

To plan the work for the SafeScrum® sprints, the company in question uses Jira and a Scrum board—see Fig. 13.4. The selected user stories are put up on the task board.

A short description of the Scrum board—also called a task board—is copied from the Agile Alliance and edited for this book [1]: *"In its most basic form, a task board can be drawn on a whiteboard or even a section of wall. The board is divided into three columns labeled "To Do", "In Progress" and "Done". Sticky notes or index cards, one for each task the team is working on, are placed in the columns reflecting the status of the tasks. Different layouts can be used, for instance, by rows instead of columns. The number and headings of the columns can vary, further columns are often used for instance to represent an activity, such as "In Test"".*

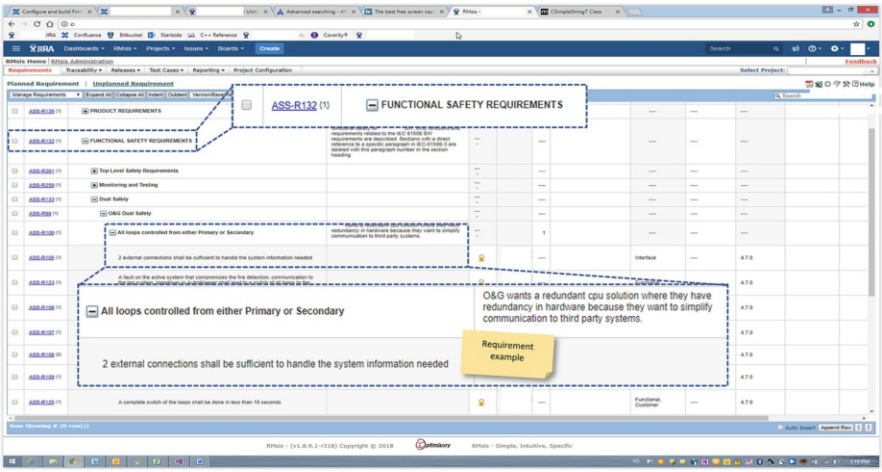

Fig. 13.2 System requirements—dual safety—controlled by RMsis

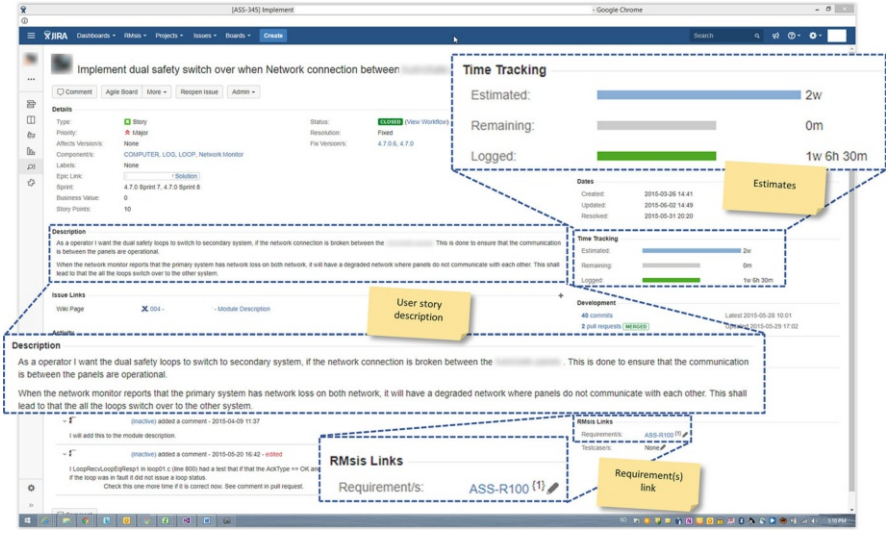

Fig. 13.3 A dual safety user story

"The task board is updated frequently, most commonly during the daily stand-up meeting, based on the team's progress since the last update. The board is commonly "reset" at the beginning of each iteration to reflect the iteration plan. Some of the expected benefits are:

- *The task board is an "information radiator"—it ensures efficient diffusion of information relevant to the whole team.*
- *The task board serves as a focal point for the daily meeting, keeping it focused on progress and obstacles".*

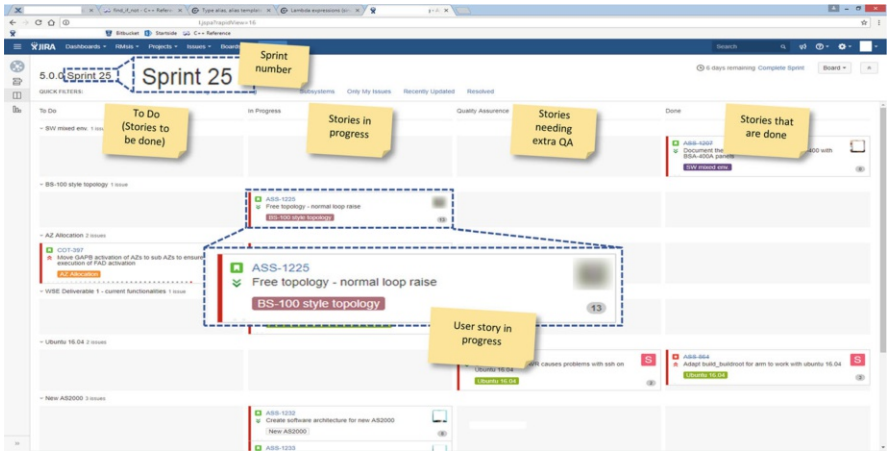

Fig. 13.4 The Scrum board

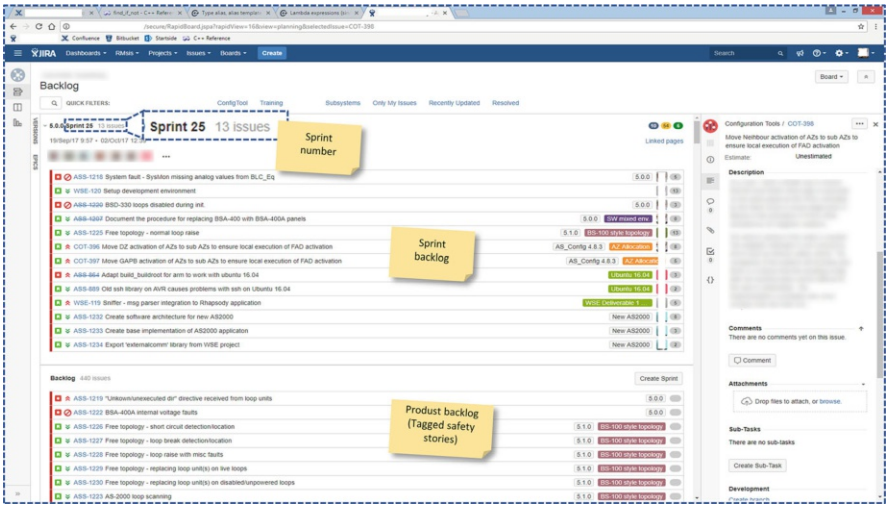

Fig. 13.5 Part of the project backlog

The simplicity and flexibility of the task board and its elementary materials (sticky notes, sticky dots, etc.) allow the team to represent any relevant information.

We start with the Scrum board shown in Fig. 13.4, where jobs are organized into four groups: "To Do", "In Progress", "Quality Assurance" and "Done". The setup of the Scrum board can be configured in each case—this is just an example fitted for one specific case/project.

The next screen-shot (Fig. 13.5) shows the sprint backlog, which, among other things contains a short description of the job to be done—for example, "set-up development environment". The colour codes (red, green, etc.) indicate the entry

type. Red indicates a bug, green indicates a story and so on. Jira uses the term "issue", which may be any job that has to be done, for example, a story. Please see Jira documentation [2] for details and terminology. On the right-hand side of the screen shot, we see the user story—"As a user I want to…"—followed by a remark—"We need to see if this is really needed…".

13.3 The Workflow

The workflow part starts with taking a story from the sprint backlog, creating a new branch and start coding. Using Jira means that user stories need to be detailed into tasks (or "sub task" as it is called in Jira)—this is done in the Sprint planning meeting. When the job is done, we need to insert the result into the relevant repository. Thus, the next step in the process is the pull request. According to [5]: *A pull request is a method of submitting contributions to an open development project. It is often the preferred way of submitting contributions to a project using a distributed version control system (DVCS) such as GIT.*

According to the process shown in Fig. 13.1, the next step is the code review—in "our" company a peer review. For this special case, the pull request screen shows that the first version of the code used in the pull request was unapproved by one of the reviewers. Then the author added a clarifying comment and the second reviewer OK'd the pull request. The comments can also be seen between the two code fragments in Fig. 13.6. In SafeScrum®, pull requests are used to ensure additional

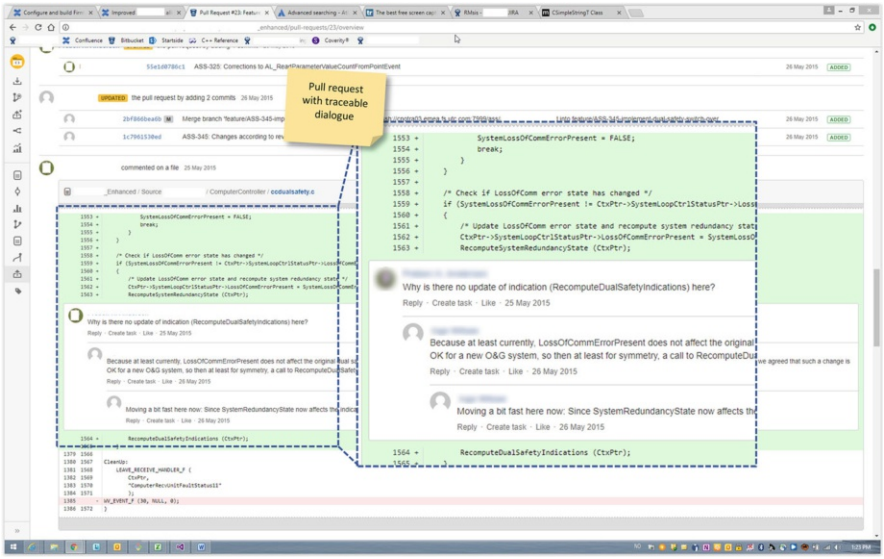

Fig. 13.6 Pull request

Metric	STFNC	STTPP	STVAR	STCCA	STCCB	STCCC	STOPT	STM21	STM20
Value	20	227	33	5625	3288	2269	29	374	202
Metric	STCDN	STTLN	STTOT	STM22	STM28	STOPN	STECT		
Value	0.690	111	579	19	19	43	0		

HIC++	Rule 1	Rule 3	Rule 2	Rule 5
Violations	5	12	3	10
Density	0.0137	0.0329	0.0082	0.0274
Namecheck	Rule 1	Rule 3	Rule 2	Rule 5
Violations	5	12	3	10
Density	0.0137	0.0329	0.0082	0.0274
QA-C++	Rule 1	Rule 3	Rule 2	Rule 5
Violations	5	12	3	10
Density	0.0137	0.0329	0.0082	0.0274

Fig. 13.7 QAC metrics summary

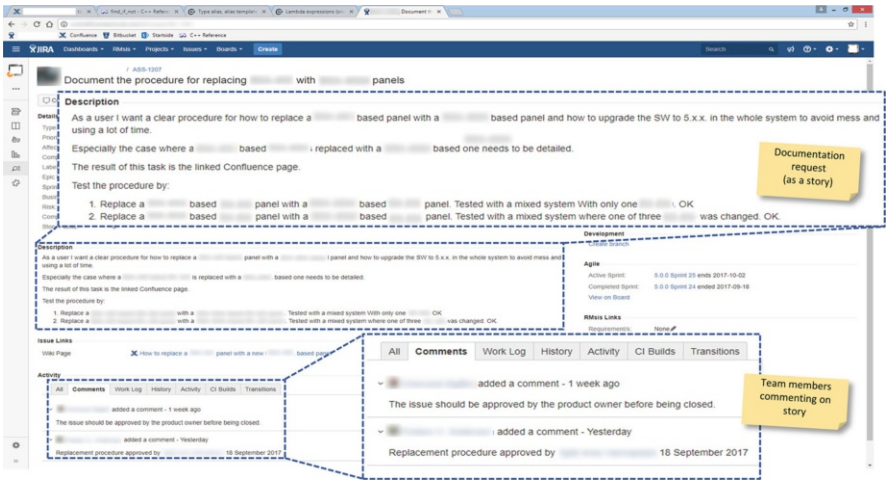

Fig. 13.8 Screen-shot of a documentation request

quality and the traceability of this process may be tracked by, for example, Bitbucket [3] (also an Atlassian tool that integrates with Jira).

An important part of the quality control is the software metrics, supplied by the QAC tool [6]. An example is shown in Fig. 13.7. Before moving on to the review meeting, each part of the code needs to be carefully checked for quality. See Sect. 7.7.1. This is just a random example with metrics that has been selected or defined specifically for this case. This is usually done as part of defining the coding standard. See Sect. 7.7.1 for details on this.

The next screen-shot, Fig. 13.8, shows the handling of a documentation request—how to handle a change of panel for a fire alarm system. Note that even though this is not a need for implementation, it is still a user story.

The user story, together with the acceptance criteria, is placed under the heading "Description". There is also an issue link to a Wiki page (in this case an internal Confluence page. Confluence is a wiki-based documentation tool—also in the Atlassian family [4]).

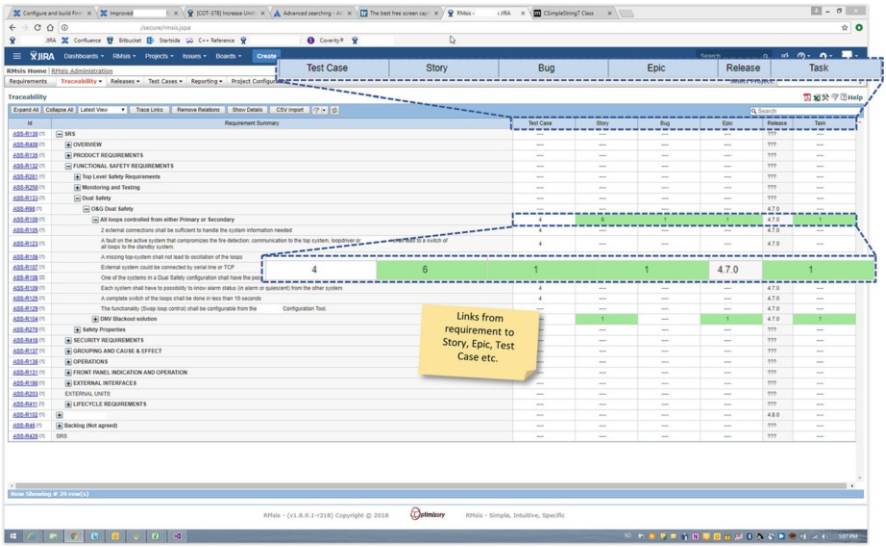

Fig. 13.9 Trace information

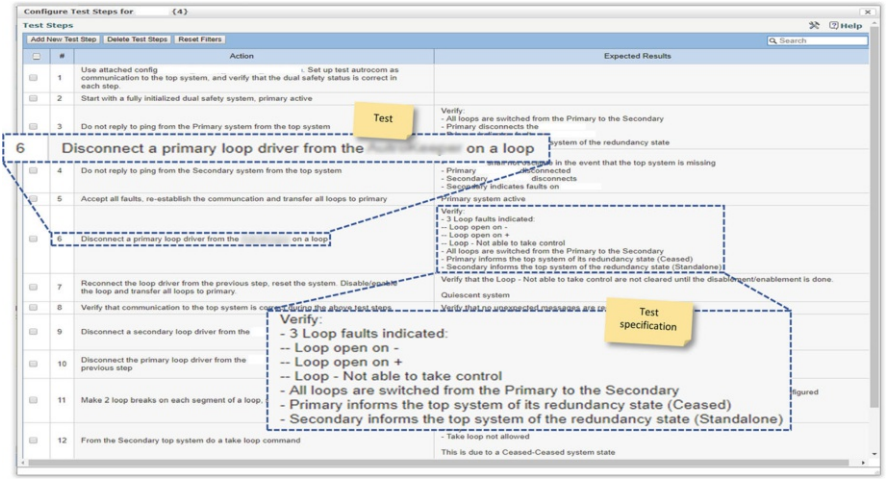

Fig. 13.10 Part of a test description

It is important to keep all trace-information up-to-date. Our company uses RMsis also for this. The screen dump in Fig. 13.9 shows a functional safety requirement, plus relevant test cases, the epic and the story, any bugs, and unfinished tasks.

In order for a test to be valuable, we need to describe each step—action, what should happen and what should be verified—expected results. Fig. 13.10 shows a typical example.

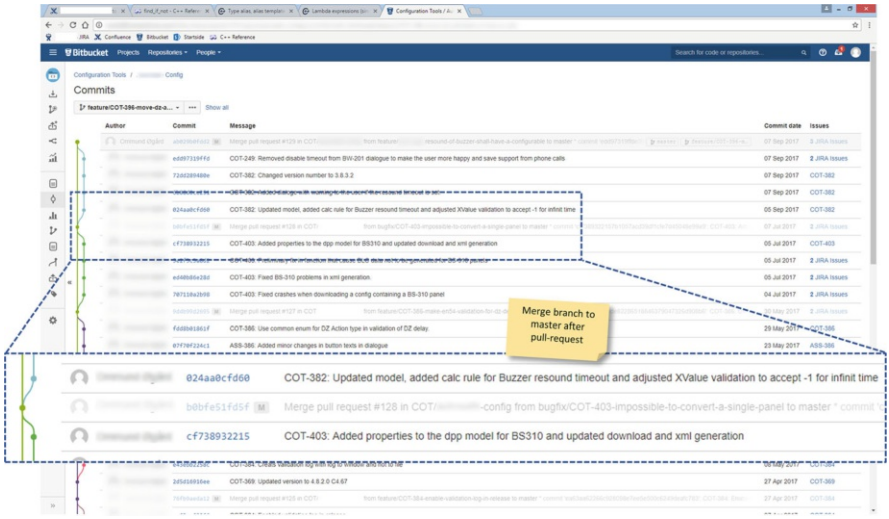

Fig. 13.11 Screen-shot from the configuration tool

13.4 Sprint Review Meeting

Our final example is from the configuration tool (Bitbucket in this case)—see Fig. 13.11—showing a series of commits. Each commit has a date, the person responsible and an explanation—for example "changed version number to 3.6.5.2".

After the first four commits, there is a "Merge pull requests" action and then the same again after the next four.

References

1. Alliance, A. (2018). *Task board*. Retrieved May 2018, from https://www.agilealliance.org/glossary/taskboard
2. Atlassian. (2018). *JIRA documentation*. Retrieved May 2018, from https://confluence.atlassian.com/jira/jira-documentation-1556.html
3. Atlassian. (2018). *Git, your way*. Retrieved May 2018, from https://bitbucket.org/product/features
4. Atlassian. (2018). *Confluence cloud documentation home*. Retrieved May 2018, from https://confluence.atlassian.com/confcloud/confluence-cloud-documentation-home-941614888.html
5. OSSWATCH. (2018). *What is version control? Why is it important for due diligence?* Retrieved May 2018, from http://oss-watch.ac.uk/resources/versioncontrol
6. PRQA. (2018). *Source code analysis with unsurpassed accuracy*. Retrieved May 2018, from http://www.prqa.com

Annexes A–D

This annex has four main parts:

- *(A) The necessary documentation needed to claim compliance with IEC 61508:2010*
- *(B) A short introduction to safety analysis in SafeScrum®, plus an overview of some important safety analysis methods, including some examples*
- *(C) Some useful UML diagrams*
- *(D) An overview of the analyses required by the IEC 61508:2010*

Annex A: Necessary Documentation

In Table 1, we have classified the documents that are specified in Table A.3 regarding software in IEC 61508-1:2010. The documents are presented in the sequence as presented in the standard. Documents may have various forms—it is however the content that matters, not the format.

There are several levels of documentation in a software project. The documents at these levels have different sources, different costs but often the same roles, both in the project itself and when it comes to certification. It is especially important in agile projects that the documents are reusable. Since we also aim for less documentation, we may combine several documents. This is especially important for small projects. From an agile point of view, the best solution is automatically generated documents as most of the software engineers wish to write code, not documents.

- **Reusable documents**—Low extra costs. These are documents where large parts are reused as is, while small parts need to be adapted for each project and even for each sprint for some documents. If reuse is the goal right from the start, the changes between projects or iterations will be small. For further information about reuse see IEEE std. 1517 "Standard for information technology—System and software life cycle processes—Reuse processes". Ed. 2 (2010).

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8

Table 1 IEC 61508-1:2010 Table A.3 regarding software documentation and corresponding classification

IEC 61508-1:2010, Table A.3 for SW	Classification According to Chap. 9 and Comments
1. Specification (software safety requirements, comprising of software safety functions requirements and software safety integrity requirements)	Generated from, e.g., a requirement management tool and/or backlog management tool and is reusable For further information see IEEE Std. 830-1998 and IEEE Std 1233-1998
2. Plan (software safety validation)	Reusable. The document can be combined with document 26 For further information see IEEE Std. 730-2002
3. Description (software architecture design)	Reusable. For further information, see ISO/IEC/IEEE Std. 42010:2011, IEEE Std. 1016:2009 and www.sysmlforum.com/ regarding SysML model management
4. Specification (software architecture integration tests)	Reusable. The standard ISO/IEC/IEEE 29119-3:2013 "Test Documentation" includes relevant information related to specification of tests
5. Specification (programmable electronic hardware and software integration tests)	Reusable. Observe the IEC 61508-4:2010 definition 3.8.1 regarding "verification" in part 4 related to integration tests. As a comment it is stated: *integration tests performed where different parts of a system are put together in a step-by-step manner and by the performance of environmental tests to ensure that all the parts work together in the specified manner*.
6. Instructions (development tools and coding manual)	Reusable. New development tools have to have relevant instructions. See existing coding manuals/information issued by Exida for C/C++ and a guideline issued by MISRA for C++. See www.misra-cpp.com/ for further information
7. Description (software system design)	Reusable. For further information, see IEEE Std. 1016:2009 "Recommended Practice for Software Design Descriptions"
8. Specification (software system integration tests)	Reusable. The document can be combined with documents 9 and 10
9. Specification (software module design)	Reusable. The document can be combined with documents 8 and 10.For further information, see IEEE Std. 1016
10. Specification (software module tests)	Reusable. Can be combined with documents 8 and 9
11. List (source code)	Generated. Source code can easily be generated directly from the code management system. In addition, many tools may automatically produce code documentations. For example, Doxygen (www.doxygen.org) and other similar tools
12. SW module design report (software module tests)	Generated. Some of the tests are generated automatically, others are semi-automatic and some are manually. ISO/IEC/IEEE 29119-

(continued)

Table 1 (continued)

IEC 61508-1:2010, Table A.3 for SW	Classification According to Chap. 9 and Comments
	3:2013 includes procedures and templates for: • Test status report • Test completion report • Test data readiness report • Test environment readiness report • Test incident report • Test completion report
13. Report (code review)	Combined. Documents 13, 14, 15, 16 and 17 can be one report. The documents can be developed gradually. There exist several tools for static code analysis (e.g. http://cppcheck.sourceforge.net/ for static C/C++ code analysis) and code review (e.g. www.parasoft.com/cpptest) See also IEEE 1028:2008, IEEE Standard for software reviews and audits. This standard defines five types of software review and audits. In this edition of the standard, there is a clear progression in informality from the most formal, audits, followed by management and technical review, to the less formal inspections, and finishing with the least formal inspection process (walk-throughs)
14. SW module testing report (software module tests)	Generated. Documents 13, 14, 15, 16 and 17 can be one report Some of the tests are generated automatically, others are semi-automatic and some are manually
15. Report (software module integration tests)	Generated. Documents 13, 14, 15, 16 and 17 can be one report Some of the tests are generated automatically, others are semi-automatic and some are manually
16. Report (software system integration tests)	Generated. Documents 13, 14, 15, 16 and 17 can be one report Some of the tests are generated automatically, others are semi-automatic and some are manually
17. Report (software architecture integration tests)	Generated. Documents 13, 14, 15, 16 and 17 can be one report Some of the tests are generated automatically, others are semi-automatic and some are manually
18. Report (programmable electronic hardware and software integration tests)	Generated. Some of the tests are generated automatically, others are semi-automatic and some are manually
19. Instructions (user)	Reusable. Can be combined with 20. For further information, see IEEE Std. 1063 and ISO/IEC/IEEE 26515:2011 “Systems and

(continued)

Table 1 (continued)

IEC 61508-1:2010, Table A.3 for SW	Classification According to Chap. 9 and Comments
	Software engineering—Developing user documentation in an agile environment"
20. Instructions (operation and maintenance)	Reusable Can be combined with document 19
21. Report (software safety validation).	Newly developed. See also Table F.7 "Software aspects of system safety validation" in IEC 61508-7:2010
22. Instructions (software modification procedures)	Reusable. See, for example, "Change impact analysis as required by safety standards, what to do?" [5]
23. Request (software modification)	Newly developed. Can be combined with document or tool as mentioned in 25
24. Report (software modification impact analysis)	Newly developed. A template has been presented in [6]
25. Log (software modification)	Newly developed. Tools exist for software modifications, for example, the open source tool Bugzilla, www.bugzilla.org. Can be combined with document 23
26. Plan (software safety)	Reusable. The document can be combined with document 2. For further information, see IEEE Std. 1228:1994 "Standard for Software Safety Plans" [4] and "The Agile safety plan" [7]
27. Plan (software verification)	Reusable
28. Report (software verification)	Generated. Some of the tests are generated automatically, others are semi-automatic and some are manually
29. Plan (software functional safety assessment)	Reusable
30. Report (software functional safety assessment)	Reusable. Finished after the last test/verification/validation report
31. Safety manual for compliant items	Reusable. May have a few remaining parts after the last test/verification/validation report

- **Combined documents**—Identify documents that can be combined into one document.
- **Automatically generated documents**—High initial costs but later low costs. This is documents that are generated for each new project or iteration by one or more tools.
- **New documents**—High costs. These are documents that have to be written more or less from scratch for each new project but there exist templates for some important documents [6]. Some standards, such as ISO/IEC/IEEE 29119-3:2013, include procedures and templates for reports such as Test status report, Test completion report, Test data readiness report, Test environment readiness report, Test incident report, Test status report and Test completion report.

Annex B: A Short Introduction to Safety Analysis

B.1 Background

What follows is a short presentation of some of the methods used for safety analysis. Each of the methods presented will have different formats and different texts for different user groups. What we present here is just one of many ways this can be done. All of the methods presented are general—that is, they can be used on any kind of system. We will, however, focus on its use on software. There exist several publications on software-FMEA (Failure Mode and Effect Analysis), software-fault trees and so on, but there seems to be little extra to gain from such an approach. Thus, we will stick to the standard methods and not discuss the software adapted methods any further.

The methods described here, especially FMEA, FMEDA and FTA, will give input to important documents and information such as Safe Failure Fraction (SFF) evaluations, PFD/CMO (Probability of Failure on Demand / Continuous Mode of Operation) and test interval evaluations, the RAMS report (e.g. PFD/CMO evaluations and calculations), SRS, element safety manual and SAR (Safety Analysis Report). In addition, they are also important for the user manual, assessment of the effect of design changes, and service instructions.

Safe Failure Fraction–SFF

λ_S is the safe failure rate, also called the spurious trip rate, λ_{DD} is the rate of dangerous but detected failures, while λ_D is the total rate of dangerous failures.

$$\mathrm{SFF} = \frac{\lambda_S + \lambda_{DD}}{\lambda_S + \lambda_D}$$

The allowable SFF will depend on the SIL—see IEC 61508-2:2010, Sect. 7.4.4.2.

Probability of Failure on Demand: PFD

If all failure probabilities are small—as they will be for a safety-critical system—the PFD can be approximated by the following expression:

$$\mathrm{PFD} = \sum_{\text{all comp}} \mathrm{PFD}_{\text{comp } x}$$

For continuous mode of operation, the estimates become more complicated, since it will depend on the architecture—for example, it is different for a 1002 and a 2003 system. See also IEC 61508-6:2010, Tables B.3.2 and B.3.3.

Test Interval

Also known as Diagnostic test interval—is the interval between on-line tests to detect faults in a safety-related system that has a specified diagnostic coverage

B.2 Participants

The most important thing when doing a safety analysis is not the method applied, but the choice of participants. In order to do a good safety analysis, you need people with knowledge and experience about the new system or similar systems that already have been set into operation, and the environment where the system shall operate.

When the choice of participants is the most important one, you will often end up with people that are not primarily safety experts. Thus, it is important that the methods you use are simple to use and easy to learn. This should hold for all the methods suggested below. In addition, the RAMS engineer should make sure that the right knowledge and experience participate in the safety analysis.

B.3 On Safety Analysis in SafeScrum®

Safety analysis must start as soon as we get the top-level requirements with or without a high-level sketch of the system. In addition, safety analysis may be needed when the system's requirements, realization or operating environment changes. Thus, the method we need must fulfil the following important requirements:

- It must be flexible when it comes to the format and amount of input—for example, it must be able to handle component diagrams (system sketches), user stories, use case diagrams and textual use cases.
- It is important to involve customers and developers in the safety assessment process and give them the opportunity to contribute. Thus, the method must be easy to learn, understand and apply.
- Since we are operating in the agile development domain, it is important that the method is well suited for handling changes to the requirements throughout the process.

There is a lot of information available when we start to write requirements for a new system, both domain-specific information and information that is generic. The best safety requirements process will depend on the information available. We will not introduce any new methods. The methods described is just a collection of concepts, put together to help with early safety requirements and analysis.

The proposed methods will make sure that all available information is taken into account to let us have an early start on the safety analysis. We will have a closer look at the suggested methods in the next sections. The steps in the proposed safety requirements process is as shown in the diagram in Fig. 1. The method described

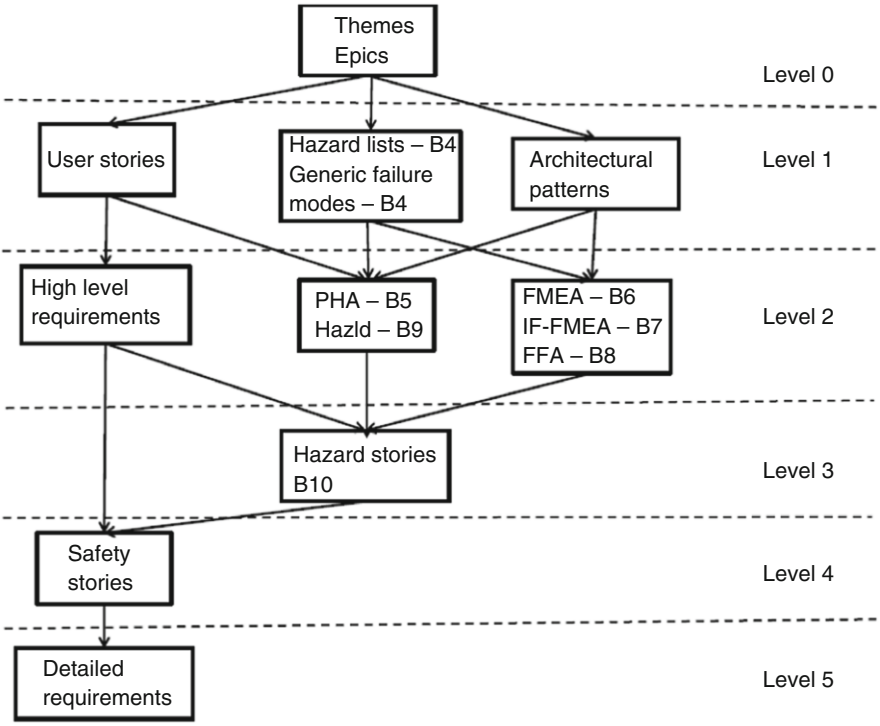

Fig. 1 Proposed requirements process

here is quite informal and includes early, informal information—for example, epics and user stories—see Sect. 6.4. In our opinion, this is necessary in order to allow all stakeholders to contribute and in order to use all available information in an efficient way. We do not necessarily need to use both FMEA, PHA and HazId. As shown later, generic failure modes—Section B.4—used in FMEA or a checklist will cover the same areas and give us the same information as a PHA or a HazId.

The process model shown in Fig. 1 has six levels—0–5. Level 0 is our starting point—the system's theme and epics. The left-hand side of the diagram contains the requirements, while the right-hand side concerns the safety analysis. Level 1 contains the process input information; level 2 contains the analysis methods that should be applied while level 3 and 4 show the high-level and detailed requirements ending with the safety stories—see Sect. 6.5.3—that provide input to the detailed safety requirements. The customer should be involved at all levels. However, this role is left out in order not to clutter the diagram.

Alternatively, the system's safety requirements are derived from the customer's perceived safety needs. There are several ways to identify the perceived safety needs. The methods that should be used must be simple, since it is important to involve all stakeholders. Thus, another alternative is common brainstorming. Beware that this

process has to be strongly managed in order not to degenerate into a process where "everything" is considered dangerous.

We will start with the top-level requirements, which people using agile development often call themes or epics. The epics are important since they describe the customer's goals. The epics also identify the application domain and the environment in which the application will operate. Thus, the epics will help us to identify the following:

- One or more relevant architectural patterns
- Domain-specific fault trees
- Domain-specific hazard lists
- User stories, which is the next level of requirements in agile development

These activities are all on level 1 in the proposed process—see Fig. 1. We might not need everything on level 1. We will need the architectural patterns so that we know the high-level components of the system but we can make do with either hazard lists or generic failure mode. The hazard list is probably more important than the generic failure modes since it is directly related to the application domain of the system.

Several domains have published their own hazard lists—also called hazard prompt lists. These are useful as a starting point for Preliminary Hazard Analysis (PHA) and for HazId, which is a simplified, brainstorming version of HazOp. A list of events that should not be allowed to happen can also be derived from domain knowledge—see, for instance, the list developed by the Federal Aviation Authority for avionics [13]. Checklists can also be used for this purpose. Each of the entries in the checklist should prompt questions like "how can this happen?" and "how can we handle it?"

The hazard lists found in the literature, on the web and so on should only be a starting point. Over time, we should make our own hazard lists so that we can add new hazards based on our own experiences. It is also advisable to include identified barriers into the hazard lists. In addition to hazard lists, domain ontologies can be of great help. For agile development, the next level of requirements after the epics is user stories. Instead of user stories, some projects will also use use-case diagram or textual use cases to detail the user stories.

We now have the information needed to start the preliminary safety analysis—shown as level 2 in Fig. 1. We will focus on FMEA—Section B.6—and PHA—Section B.5. There are several reasons for focusing on FMEA:

- As opposed to HazOp, which is surrounded by a large amount of ceremony, FMEA is easy to understand and easy to use. We just need to answer the question "What can happen if this component fails in such and such a way?"
- There exist several sets of generic failure modes for FMEA—for example, for hardware, software and wetware (operators) [11]. This makes it easy to get started.
- The method can be applied to components at all levels, sub-systems and requirements—high-level and low-level.

There exists a more efficient version of FMEA, called IF-FMEA (Input Focused FMEA)—Section B.7, which allows you to include failures caused by input from other components. We may also need to assess the Safe Failure Fraction. This can be done by using an extended version of FMEA called FMEDA—Failure Mode Effect and Detection Analysis, Section B.11. We will continue by having a quick look at

- IF-FMEA on generic components—Section B.7.
- PHA and HazId with user stories as input—Sections B.5 and B.9. Note that this analysis suggests both new requirements and new barriers.

Starting early with a safety analysis, we know that there will be changes. The reasons are legion—new needs, better understanding of the consequences of previous choices, changes in the market—the list goes on and on. The challenge is how not to introduce new hazards when something is changed. We can categorize changes into two categories:

- New requirements or changes to existing requirements. These changes stem from the customer. New requirements will go through the same safety analysis as the previous requirements, while changed requirement will be handled by the customer and developers together at the start of a sprint.
- New code or changes to existing code. Such changes stem from a sprint retrospective and will be handled by the developers as part of the planning of a new sprint.

In both cases we might need to perform a change impact analysis—see Sect. 8.2. Note that introducing new requirements usually implies that we also will have to change existing code, while changes to code do not necessarily imply changes to requirements. Important questions in these situations are:

- When we add or remove components or change existing code—what code and which requirements uses this code?
- When we add or change requirements—what code is used to realize this requirement?
- In all cases, which safety analyses need to be rerun?

These questions can only be answered in an efficient way by using trace information. This will need to include not only the standard types of trace information—links from requirements to architecture, from architecture to design and so on—but also trace information of which safety analysis involves which requirements and which components. For all cases, we will need to write a Change Analysis Impact Report, which is an important input for the safety assessor.

Experience from our industrial partners shows that developers in a company with a strong safety culture will be able to perform the necessary safety analysis themselves in most cases. In some special cases, we will need to involve other personnel to supply extra domain knowledge. The change processes can be described as follows:

- A new requirement—perform a new hazard analysis, focusing on how the requirement can fail and what the consequences are. Will any of the existing barriers take care of these consequences or do we need new ones? Will the requirement create a need for new requirements related to safety? Note that new requirements also might create a need for new components.
- For a changed requirement, we need to look at the safety analysis done for the original requirement—especially the assumptions that are used. Will the changes create needs for a new safety analysis?
- Changes to existing code. Traces will identify which requirements the code supports and thus enable us to see if the changes will influence current safety or introduce new safety threats.

The important message is that by using all available sources of information of a system, it is possible to get an early start of safety analysis. We have three specific conclusions and one recommendation.

- By using existing generic and domain-specific information it is possible to get an early start on safety analysis. This is important since architectural decisions made early in a project—agile or not—are expensive to change later.
- Architectural and problem patterns are important in order to identify generic components that can be analysed using variations of FMEA.
- FMEA and its variant IF-FMEA work well in an agile setting.

We recommend companies developing safety-critical systems to build and maintain a library of relevant patterns, hazards, barriers, generic failure modes and generic fault trees.

B.4 Probability and Consequences

Most safety analysis methods will need some kind of assessment for probability and consequence (severity) of failures. For most hardware components, the probability of the failure modes will be available from the component fact sheet, while the consequences, depending on the environment, are not all that obvious. In the absence of data that can be used to estimate failure probability and consequences, we have to use qualitative assessments. Some examples of such assessment scales are ISO 13849:2015 with two levels and IEC 62061:2005 with five levels. Three levels is an OK alternative for both consequences and probability. The important thing is to describe how to select the right level. The following is a simple example for occurrence probability assessment. A similar set of grading can be defined for consequences.

- LOW: has rarely been a problem and never occurred for this type of systems
- MEDIUM: will most likely occur for this type of systems
- HIGH: will occur for this type of system, and has occurred in the past

Table 2 can now be used to assess the risk.

Table 2 Risk assessment table

		Consequences/severity		
		H	M	L
Probability	H	H	H	M
	M	H	M	L
	L	M	L	L

The risks in the L-area of the table are OK, the M's should be dealt with if possible, while the H-area definitively is a no-go area—risks that wind up here must be dealt with.

B.5 Generic Failure Modes and Hazard Lists

First and foremost—a failure mode is not a fault but a word or sentence that describes how a component or system can fail. Failure modes are used to identify failure causes and effects. A generic failure mode is thus a generic description of how a component or system can fail. Thus, generic failure modes must be connected to a system or component and related to an environment. Generic failure modes can thus also be used as cue words. In both cases, the connection between the generic failure mode and the effect on the environment must be done by people who understand how the system or component works and how its behaviour affects its environment. An example, taken from NRC—Nuclear Regulatory Commission [8] is shown in Table 3.

A hazard list must be domain specific in order to be useful. Some hazard lists are combined with a list of relevant situations. Table 4 shows a hazard list for automobiles, taken from [11].

A hazard list is a good overview of possible high-level hazards and a good starting point for further analysis.

B.6 PHA: Preliminary Hazard Analysis

PHA has a lot in common with HazId (see below). It is, however, handled separately here since the term PHA is quite common in the literature. The work sheet for PHA is shown below (Table 5).

We are talking of *potential* accidents. Thus, a list of earlier accidents and near misses in this area will be of great help. However, as in all safety analyses, the best way to get good results is to have competent people. The main effects column links the potential accident to the system's operating environment while corrective or preventive measures will be important input to those who shall write the safety requirements. It is also an important input to the next steps in the process, such as the

Table 3 Example of generic failure modes

ID	Failure mode	Elaboration	Remarks
A1	Fail to perform the function at the required time	Deviation from requirement in time domain	Omission, No action, No output, Reacts too late
A2	Fail to perform the function with correct value	Deviation from requirement in value domain	Wrong output
A3	Performance of an unwanted function	Deviation from expected performance	Commission, Wrong action
A4	Interference or unexpected coupling with another module	Deviation from expected system performance due to module interaction	Commission

Table 4 Generic hazard for automobiles

ID	Hazard
1	Unintended start
2	Start opposite to the intended direction of travel
3	Unintended accelerations
4	Unintended deceleration
5	Loss of steering power
6	Failure of the braking system
7	Electric shock
8	Fire

Table 5 The PHA worksheet

Hazard – potential accident	Causes	Main effects	Accident severity category	Corrective or preventive measures

HazId and the FMEA. Note that many companies skip the PHA altogether and go directly to the HazId.

B.7 FMEA: Failure Mode and Effect Analysis

FMEA can be used throughout the development process. The process is simple:

- Set up an analysis group. It is important that people with knowledge of the system-to-be and its operating environment participate.

Table 6 FMEA work sheet—early phases

Unit description	Failure description		Failure effect on the next level	Recommendations
	Failure mode	Failure cause		

- Consider each identified component—the system, its sub-systems or individual components and ask: "How can this element fail—failure mode—and what will be the consequences—failure effect?"
- Try to identify possible causes. This is important for the recommendations on what to do with the failure.

From an agile viewpoint it is important to remember that as our experience increases, for example, from daily stand-ups and sprint reviews, it is important to update the FMEA worksheet, just as we update any other information used in the system development.

Some purists will claim that it can only be used on the finished system—when all the components have been identified. However, practical experience has shown that is can be used in any phase of product development as long as we can identify components or functionality that can fail. The basic version will use a table similar to the one shown in Table 6. This table should only be used at an early stage of analysis. Note that the failure causes will always be important when we want to identify case–consequence chains. The recommendations are the important part since this is used to improve the system design.

We can use this format, for example, to do an FMEA based on user stories. The following is a small example, based on one of the user stories for a distributed fire alarm system (Tables 7 and 8). This fire alarm system consists of a set of smoke and heat detectors placed at strategic places in the building, a set of acoustic alarm devices, and a central control unit.

When we have more information available, we can add more details to the FMEA table, as shown in the next table (Table 9). In the later phases, it is also practical to include such factors as how to detect the failure. The method is then often called FMEDA—Failure Mode Effect and Diagnostics Analysis (see next section).

The FMEA can be more efficient if we use a set of predefined, domain-specific failure modes.

At this early stage, where we just have a simple diagram, we should use the FMEA for early project phases (Table 10).

At the next level of development and analysis, we add more details, in this case, details of the inner working of the controller unit.

The FMEA for the "Set temperature" part of this detailed system description can, for example, be as shown in the Table 11. Note that "Set temperature" takes input from the "Temp versus Time" table and sends the temperature information to the control software.

Table 7 Example user story

User story ID:	Local alarm
As a	House owner
I want	To be made aware of the fire
So that	I can start necessary actions—for example, call the fire brigade

Table 8 FMEA based on the user story in Table 7

User story ID	Failure description		Failure effect on the next level	Recommendations
	Failure mode	Failure cause		
Local alarm	No alarm	Loss of power Broken controller connection No input to central controller	No call to fire brigade	Ping on connection
	Wrong alarm	Central controller error	Unnecessary call to fire brigade	Require SIL 2 software
	Too late alarm	Central controller error	Too late call to fire brigade	SIL 2 software

Table 9 FMEA work sheet—late phases

Unit description		**Failure description**			**Failure effect**		**Recommendations**
Function	**Operational conditions**	**Failure mode**	**Failure cause**	**Failure detection**	**On other units**	**On the system's functionality**	

Table 10 Example of FMEA—early phases

Unit description	Failure description		Failure effect	Recommendations
	Failure mode	Failure cause		
Controller	No action	Power loss	No heat	Feedback switch info to controller for verification
	Wrong action	Switch failure—for example, switch stuck	Incorrect temperature control	
	Too late action	Delayed switch change	Delayed temperature control	

Table 11 Example of FMEA—later phases

Unit description		Failure description			Failure effect		
Function	Operational conditions	Failure mode	Failure cause	Failure detection	On other units	On the system's functionality	Recommendations
Set temp	Normal	Wrong read	SW error	Check for reasonable values	Wrong info to control	Wrong temp at defined time	Remark 1
		Wrong output					Remark 1
		Wrong data	Wrong input				Manual check of Temp table

Remark 1: If an unreasonable value is detected, the component should keep the current temperature setting and give an audio alarm.

B.8 IF-FMEA: Input Focused FMEA

The idea is to add input from other sources to the set of failure causes. If we use the FMEA from the heat element controller in Figs. 2 and 3, the IF-FMEA will have an extra column under the heading "Failure description", as shown in Table 12.

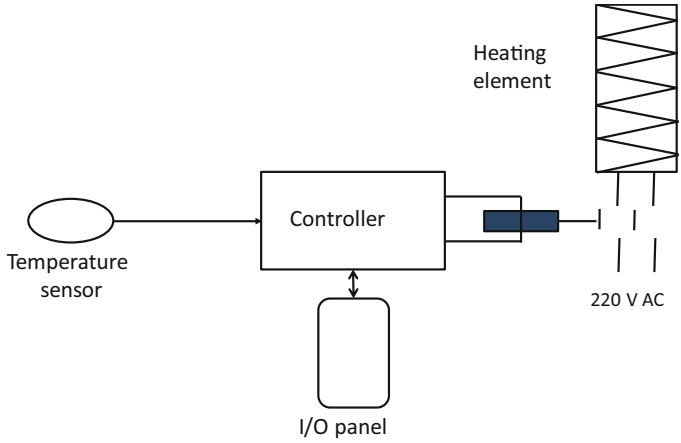

Fig. 2 Heating element controller

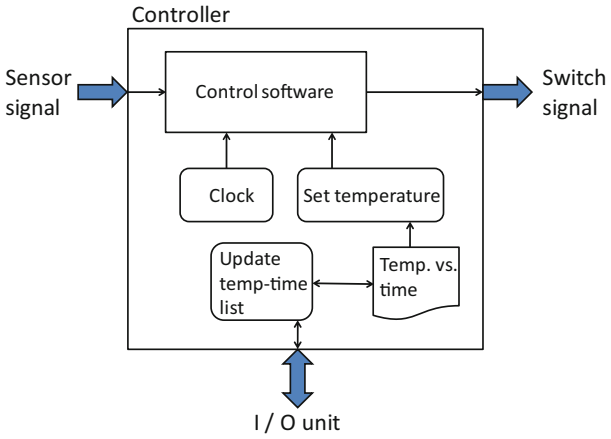

Fig. 3 Temperature controller details

Table 12 Example of IF-FMEA—later phases

Unit description	Failure description			Failure effect	Recommendations
	Failure mode	Component failure cause	Input failure cause		
Controller	No action	Power loss	No signal from sensor	No heat	Feedback switch info to controller for verification
	Wrong action	Switch failure—for example, switch stuck	Wrong signal from sensor	Incorrect temperature control	
	Too late action	Delayed switch change	–	Delayed temperature control	

Combining an input controller and barriers against illegal or unreasonable input will help to protect against some of the input errors in the example above. Other mechanisms, such as a ping protocol, will help to defend the system against "No signal from sensor".

B.9 FFA: Functional Failure Analysis

While a component-based FMEA focuses on the components of a system, the FFA focuses on its functions. The analysis process will consist of taking each system function and applying each functional failure mode—generic or specific. Not all failure modes will be relevant for all functions. The table for a functional FMEA is shown in Table 13.

S. Burge recommends the analyst to use the generic failure modes in the functional FFA as guidewords [2]. We may, for instance, use the generic functional failure modes "Over", "Under", "No", "Intermittent" and "Unintended". Using a term such as "Over" as a guideword we do as follows:

1. Select a function.
2. Apply the generic failure modes to each function. The two following examples will show how it works for the function "Read temperature".
   (a) If the generic failure mode is "Over", we should look at the consequences of too high a temperature.
   (b) If the generic failure mode is "No", we should look at the consequence of no result from the reading.

Table 13 FFA work sheet

Function	**Function description**			
Functional failure mode	**Effects**	**Cause**	**Detection** / **Current method**	**Comments**

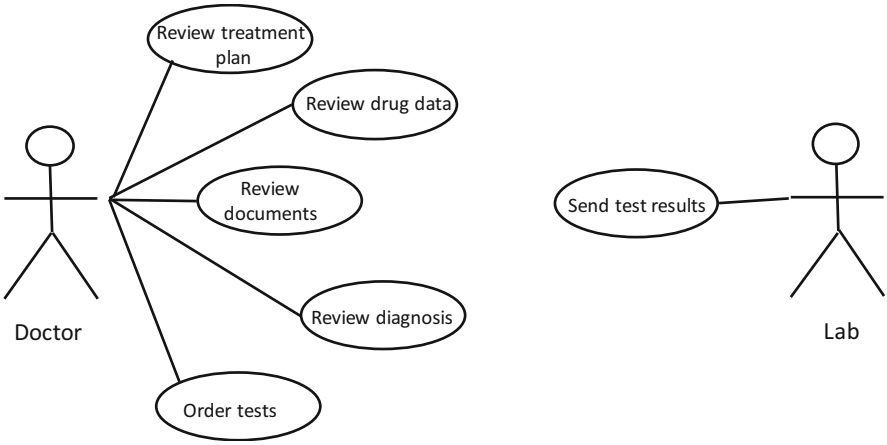

Fig. 4 Hospital use case

3. If the consequences of the failure mode are severe, we need to look for a detection method so that the error can be handled. The solution will be inserted as a new requirement.

Note that not all generic failure modes might make sense in all cases.

The FFA approach works well for instance together with a graphical use case diagram. In this case, we can apply the FFA to each function ("bubble") in the diagram and describe the consequences if this function fails to deliver the specified functionality (Fig. 4).

The use case in Fig. 4 has six functions. Each of these can be analysed using FFA. We will just look at one of them—order tests (Table 14).

B.10 HazId: Hazard Identification

HazId is an alternative to HazOp (Hazard and Operability study), especially in the early phases. Experience has shown that HazId works especially well as a medium

Table 14 Example of FFA

Function Order tests	Function description The doctor orders a test from the hospital's laboratory via the internal network			
Functional failure mode	Effects	Cause	Detection Current method	Comments
No action	No order sent	Connection down	Ping connection ACK	
Too late action	Order sent too late	Traffic delay	Priority scheme	
Wrong action	Wrong order sent	Error in protocol stack	–	
		Error in network	Test traffic	

Table 15 Simple HazId table based on functions

The system shall have an emergency stop		
Failure condition	Effect of failure	Remark
Not fulfilled	No emergency stop possible	May lead to people getting hurt

during safety brain storming sessions. It consists of two parts: (1) identify the system's components or functionality and (2) assess how each function or component's failures can influence the system's environment. Just as FMEA, its main purpose is to identify dangerous situations and events and then recommend changes—for example, barriers.

HazId Based on Functions

HazId based on system functionality is straightforward. We study each functional requirement based on the question "What will happen if this functional requirement is not satisfied?" A simple example is shown in Table 15.

This analysis can later be refined by introducing more failure conditions or more specific failure conditions for each functional requirement. Possible refinements are, for example, "Only partly fulfilled" or "Only fulfilled under certain conditions".

HazId Based on Components

The system's requirements are used as a starting point. The idea is to use the requirements to identify the system's components. In the HazId table, we will consider consequences of problems for each component. The barrier column is used to identify existing barriers. It is up to the analysis to decide whether they are sufficient for the identified problems. New suggested barriers and other design changes are inserted into the "Recommendation" column.

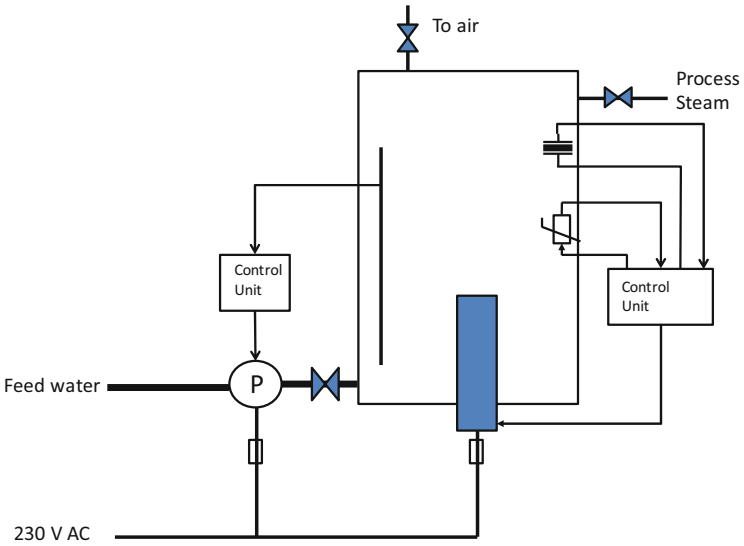

Fig. 5 Simple steam boiler control system

Table 16 Simple HazId table based on components

Product					
Component problem	Consequences	Barriers	Recommendation	Responsible	Action
Wrong pump signal	Too much water	–	Flow meter in the feed-water pipe	John	Add function in water level control unit
	Too little water	–	Alarm if water level too low	Peter	Add function in water level control unit
Wrong heating unit signal	Too much heat	Safety valve	–	Peter	–
	Too little heat	–	Alarm if too low water temperature	Tim	Add function in temperature control unit

Consider the following example—a simple steam boiler control system.

The system in Fig. 5 has two control units: one for the water level, controlling the water pump and one for the temperature and pressure, controlling the heating (Table 16).

An alternative table, HazId table, is shown in Table 16. This table requires less information. The main advantage is the use of component type, which will enable the analysis to use component-specific generic failure modes and a set of generic, high-level, failure causes (see Table 17). The concept of generic failure modes and generic

Table 17 Simple HazId table based on components—alternative

Component type	Failure mode	Failure cause	Comments
Sensor	Wrong output	Component failure	Fail high

Table 18 Component-type-specific generic failure modes

Component type	Failure mode	Failure cause	Comments
Sensor	No output	Component failure	No signal
	Wrong output		Fail high
			Fail low
Actuator	No action	Mechanical / electrical error in actuator	Stuck on
			Stuck off
	Wrong action	Mechanical / electrical error in actuator Wrong input	Acts when it should not
			Did not act when it should
Computer control system	Omission	Hardware failure Software failure Wrong input Wrong component state	Something not done
	Commission		Something more done
	Reacts too late	Wrong component state	
	Annunciated loss of function		Failure detected or diagnosed

failure cause depending on component type stems from the CESAR project [9] (Table 18).

HazId Based on Tasks

A task is a set of functions—the functions needed to perform the task. A useful format is shown below. This format focuses on the hazards and the control or barrier. In addition, the table contains the risk score before and after introducing the barrier plus the ID of the person or persons responsible for the barriers.

A task related to the system in Fig. 5 could, for instance, be: "Keep the water level within defined upper and lower limits". Hazards could, for example, be related to feed-water availability, feeding pump, non-return valve and water level indicator (Table 19).

B.11 Hazard Stories

The process for developing hazard stories is a brainstorming process and has the five steps shown below. Step numbers refer to the numbers on the left-hand side of the diagram shown in Fig. 6.

Table 19 Suggested HazId table based on tasks

Task description		Reference number	Review date
Hazard description		Risk score	
		Initial	Current
Control or barrier	Reference	Owner	Status

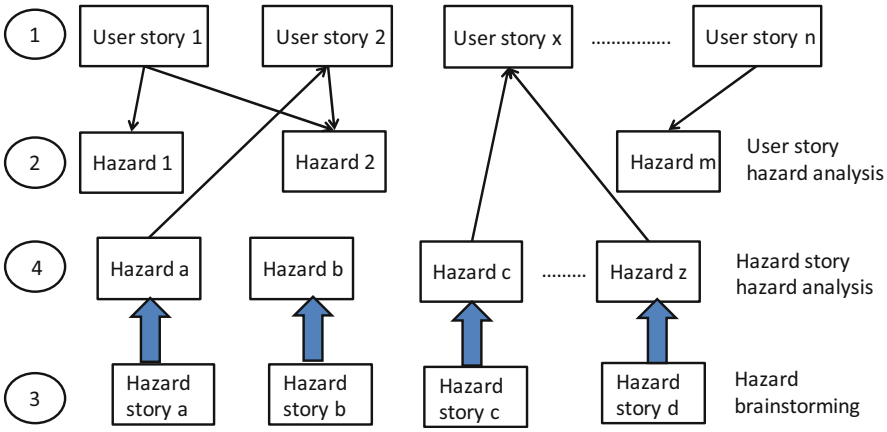

Fig. 6 Hazard analysis—four step process

1. Write down the epics and the user stories—step 1 in the diagram.
2. Do, for example, PHA or FMEA based on the user stories (see example in annex A)—step 2 in the diagram.
3. Get together users, safety experts, security experts and the product owner for a brainstorming process—step 3, part 1.
4. Put the results from the brainstorming into the Hazard story format—step 3, part 2.
5. Convert the hazard stories to hazards and update the agile hazard log—step 4.

If the hazard stories bring up the need for new requirements we should update the user stories or add new ones. In addition, we also need to update the SRS. The people involved in the brainstorming process are the members of the team and they should already know the user stories and the epics. If necessary, domain or safety experts can be included. All participants will have access to the results from the first process, based on the user stories and they should get to know them beforehand in order to

reduce the more obvious ideas. Even though people might feel that there is nothing to add, it can make them dig deeper into their imagination. A case study [12] indicates that when the "mechanical" hazard stories were ready, the participants might be more creative. They might improve their hazard stories based on the ideas from the session, if their stories are too complicated or unclear.

B.12 FMEDA: Failure Mode Effect and Diagnostics Analysis

FMEDA is an extension of FMEA and is normally used for hardware and its main purpose is to find the diagnostic coverage for the system. The information can be organized as shown in Table 20. The total failure rate can be obtained from product data sheets, while the rates for safe and dangerous failures, respectively, will depend on how the component is used and is often decided using engineers judgement. The "detected rate" will depend on how we instrument the system—see Table 21. See also IEC 61508-7:2010, annex A for concrete techniques.

The information needed is shown in the following list:

- Component information, including brand and make. This information is taken from the product fact sheet.
- Failure modes: we recommend using generic failure modes, for example, the ones used in Table 3. See also [8].

Table 20 FMEDA Work Sheet for Hardware

Component information	Failure modes	Effect	FIT	Failures			Diagnostic method
				Safe	Dangerous	Detected	
Total number of FIT							
Failure rates							

Table 21 FMEDA Example

Component information	Failure modes	Effect	FIT	Failures			Diagnostic method
				Safe	Dangerous	Detected	
Hoisting motor	Stop working	Serious	100	50	50	30	
	Wrong action	Serious	10	0	10	3	
Total number of FIT			110	50	60	33	
Failure rates			$1.1\ 10^{-6}$	$5.0\ 10^{-7}$	$6.0\ 10^{-7}$	$3.3\ 10^{-7}$	

- Effect: This can be real costs, if they can be assessed but usually a three-level scale will be sufficient—for example, high, medium and low. See also table 2.
- FIT: the sum of safe and dangerous failures. FIT: Failure In Time (1×10^{-9} failures per hour).
- Failures—the expected number of safe and dangerous failures in the component's useful life. The useful life time is calculated as the life time of the component, multiplied by the number of components in use. For component failure rates, see, for instance, the MIL 217, the Exida handbooks or the OREDA handbook.
- Detected is the estimated number of dangerous failures detected. The failure rates are indicated for both safe and dangerous failures. These rates will depend on the diagnostic methods used.
- Diagnostic method: Examples are watchdogs, 2oo3 architectures and hardware self-tests (e.g., walking bit).
- The failure rates are defined as Total FIT divided by the component's useful life time.

For computation of diagnostic coverage, see IEC 61508-2:2010, Annexes A, C and E. Note the notation AooB—A out of B—which means that at least A out of the B parallel components must be working in order that the system shall work. Thus, the system in Fig. 7 is a 1oo2 system—at least one of the two channels needs to be up and running.

Most of the techniques used to discover errors have, up till now, been hardware-oriented. There is, however, a trend towards more use of software to diagnose hardware during operation. The figure below shows the diagnostic components for a two-channel system. If the error is hardware related, a 1oo2 solution is really a 2oo2 solution and actually makes the system less reliable since we now have doubled the number of hardware components that may fail. However, if the errors are software related the diagnostics might pick them up. This goes for errors that manifest themselves as values outside reasonable ranges, too long response time and connection errors to, for example, sensors or actuators. Software watchdogs and the ping protocol are but two examples of diagnostic methods that can be used.

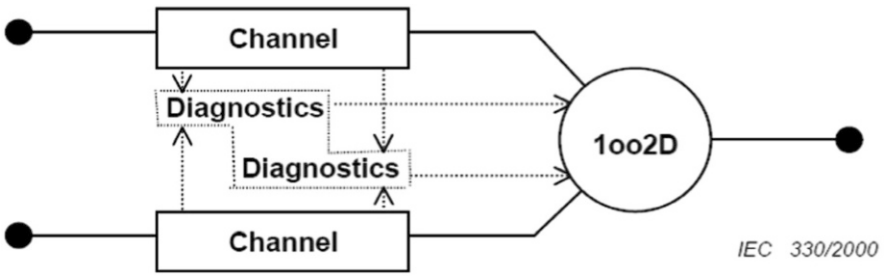

Fig. 7 Example of the use of a diagnostic unit

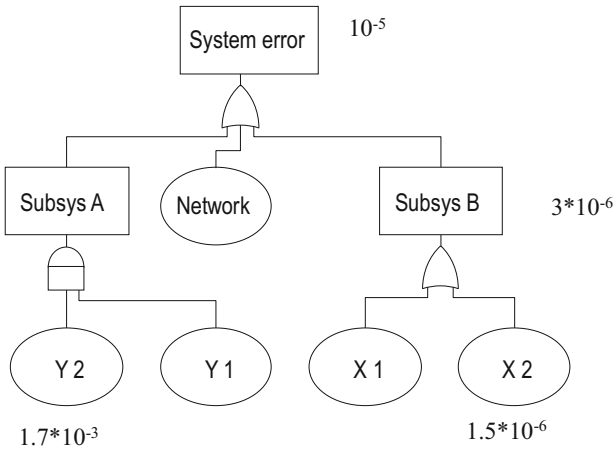

Fig. 8 Simple fault tree

A simple example of computing the safe failure fraction is shown below. We assume that there will be sold 1000 units and expected (guaranteed) life time is 10 years which gives us approximately 10^8 hours useful life time.

B.13 FTA: Fault Tree Analysis

FTA is a straight top-down method. It focuses on events, not on system structure or system requirements. The method is used to understand causes and effects of events in system components and how the events can combine to create the top event. The FTA uses a large set of diagrammatic symbols to create a map of how the events combine. The diagram in Fig. 8 shows the four most important symbols.

The symbols used in the above diagram are:

- A rectangle—an event that will be further broken down into more details.
- A roman arc—an AND gate. When all input events occur, the output event will occur.
- A gothic arc—an OR gate. When at least one of the input events occurs, the output event will occur.
- A circle—a basic event, that is, an event that will not be broken down any further.

Thus, the event marked "System error" will occur if event {Y1, Y2}, {Network}, {X1} or {X2} occur. The four previously mentioned sets are called cut sets. A fault tree and its cut sets can be used in at least two ways—to estimate the probability of the top event and to make design decisions. We will focus on the latter.

In general—the higher up in the tree we find OR-gates, the less reliable the system is. Another way to assess the reliability is to look at the size of the cut sets. Cut sets with only one member represent single points of failure. For the system in the diagram above, we see, for example, a failure in component X1, X2 or Network will bring the system down. A failure in component Y1 will not have any effect since

we also need component Y2 to fail in order to create a system failure. This pattern will for instance occur when we have an error-prone component guarded by a barrier.

B.14 Hazards Under No–Fault Conditions

Hazard analysis is focused on how the product, by failing, can create danger. However, there are two other sources of danger that need to be considered—dangers created by faulty installation and dangers created by wrong use of the product. These problems are not the developers' responsibility but the product should be accompanied by a note saying how the product should be installed, used and maintained if it shall remain safe. A good example is the EU Commission Regulation No. 347/2012-3.4.1 [1], which states:

"The manufacturer shall provide a statement which affirms that the strategy chosen to achieve the System's objectives will not, under non-fault conditions, prejudice the safe operation of systems which are subject to the provisions of this regulation."

In order to provide this or a relating statement, we need to identify the hazards that can materialize due to errors in use or installation. An efficient way to identify non-fault dangers is to use a brainstorming process and fill in Table 22—[10].

In the example above, we see that we need to have a statement in the installation guide that specifies that the sensor-to-controller cable must be shorter that X meters; alternatively that only the enclosed cables must be used.

The outcome of this process should be a report stating the product's limitations when it comes to installation and use together with the possible hazards. It is the developers' responsibility to make the hazards clear to the customer. However, it is the customer's responsibility to decide how he or she will cope with the identified hazards.

Table 22 Analysis of non-fault hazards

Situation	Needed behaviour	Specified function	Limitations	Hazard
Too high boiler temperature	Reduced heather effect	Boiler temperature control	Sensor-to-controller cable can be maximum X meters long	Boiler overheating
. . ..	. . ..	. . ..	. . ..	. . ..
. . ..	. . ..	. . ..	. . ..	. . ..

Annex C: Useful UML Diagrams

UML is a rich language with many possibilities. The most important characteristic of UML, however, is that it can be used both formally and informally—from small idea sketches on the back of an envelope, to a rigid, formal notation in a tool. This makes the language ideal for agile and iterative processes such as SafeScrum®. The main reason for this is that when we use the UML notation, it is easy to sketch a possible solution to a problem. The proposed solution can be discussed and elaborated on, for example, on a whiteboard and, when finished, a snapshot will serve as documentation. In addition, UML diagrams are convenient for safety analysis. Both of these points will improve project communication.

One of our industrial partners uses the MVC (Model View Controller) pattern, so we will illustrate UML with class and sequence diagrams from this pattern. The class diagram for MVC is shown in Fig. 9. The symbols used are explained in Fig. 10.

The class diagram identifies several functions—for example, GetState and SetState—and variables such as "subject state" and "observer state". The class diagram is not well suited for hazard analysis since the system's communication with the environment is partly hidden. However, we can start by asking how each function can fail, what will happen if one of the variables gets a wrong value and so on. We can also use FMEA or IF-FMEA applying generic failure modes to perform safety analysis.

Navigability implies that class A is accessing information found in class B. For example, the controller updates the model in Fig. 9. Aggregation is the **part-of** relationship and implies that class B is a part of A. For example, the controller is a part of the view. The third symbol—composition—means that class A is a collection of one or more class B. There are two important differences between composition and aggregation. For composition, any instance of B can belong to only one A, and if

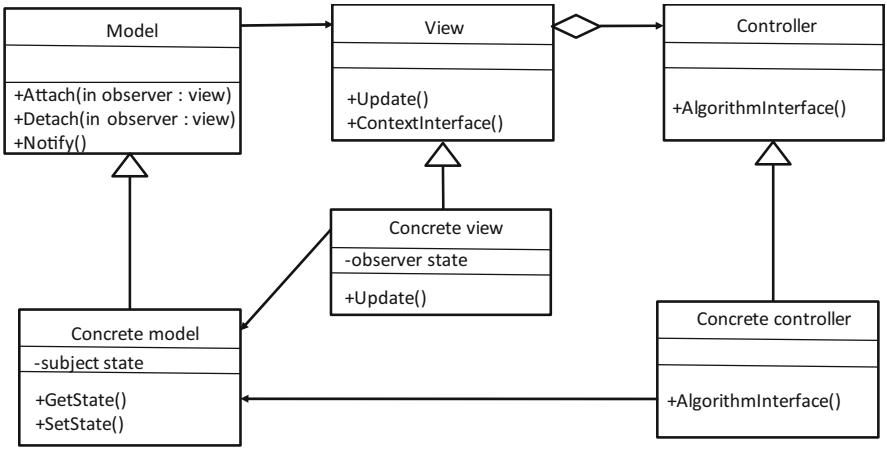

Fig. 9 Class diagram for MVC-pattern

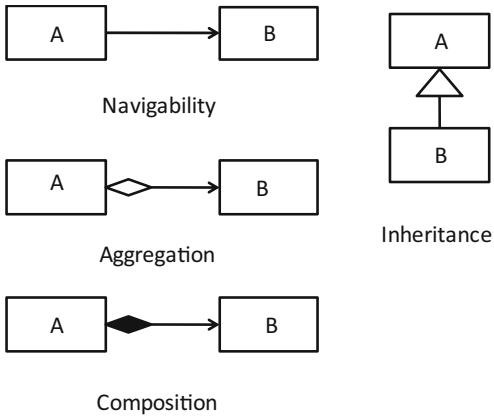

Fig. 10 Some commonly used UML-symbols

you delete A, all instances of B are also deleted. Note that several UML-experts, for example, Fowler [3]—suggest that you should not use aggregations.

The last diagram in Fig. 10, marked inheritance, shows that class B inherits all characteristics of class A. For example, "Concrete view" in Fig. 9 inherits all characteristics of the general "View". In addition, it adds some characteristics of its own, such as "observer state".

A sequence diagram is simple to make and extremely efficient to show how several objects cooperate. The most common notations are shown in Fig. 11—boxes on top are instances of classes (objects), and the vertical, narrow boxes describe the lifeline of each instance. The horizontal arrows show messages passing between the objects. There shall be an explanatory text connected to each arrow.

There are three types of arrows used in sequence diagrams—synchronous messages, asynchronous messages and returns. One feature of the sequence diagrams that is used all too seldom is the possibility to show conditions, alternatives and loops. The diagram notations are shown in Figs. 11, 12 and 13, followed by an example in Fig. 14, found in Fowler's book on UML [3]. By using these notations, we can show, in a simple way, complex algorithms in a sequence diagram (Fig. 14).

UML sequence diagrams are especially useful for safety analysis. Experiments have shown that sequence diagrams outperform other methods when it comes to identifying internal failure modes—see [8]. The reason for this is that the system's components and how these components exchange information are made easy to understand.

We can perform a safety analysis of this part of the system (dispatch handling) by asking questions such as:

- How can the "regular: Distributor" fail and what will be the consequences?
- What happens if the guard is wrongly set?
- What happens if the "Messenger" is down?

The answers to these questions will influence what we do next, such as more testing, watchdogs for one or more of the processes, or a redesign of some parts of the system.

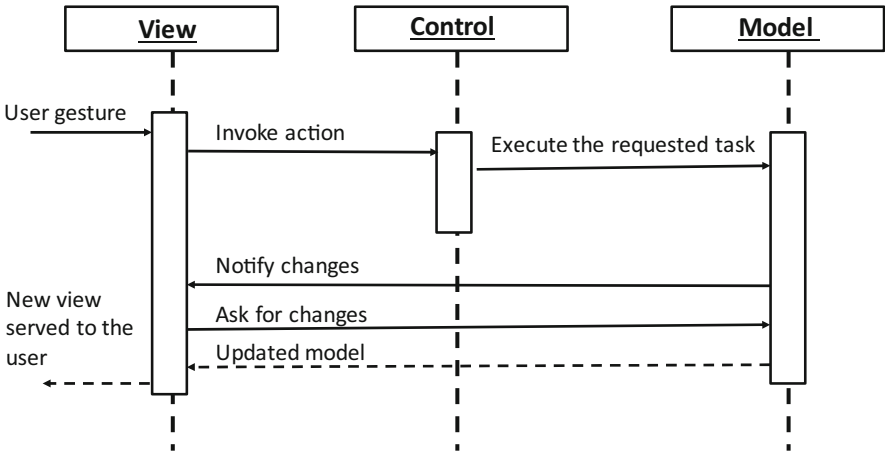

Fig. 11 Sequence diagram for MVC pattern

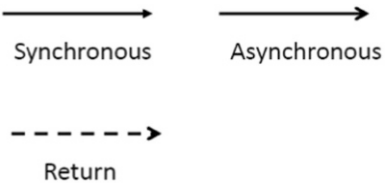

Fig. 12 Arrows used in sequence diagrams

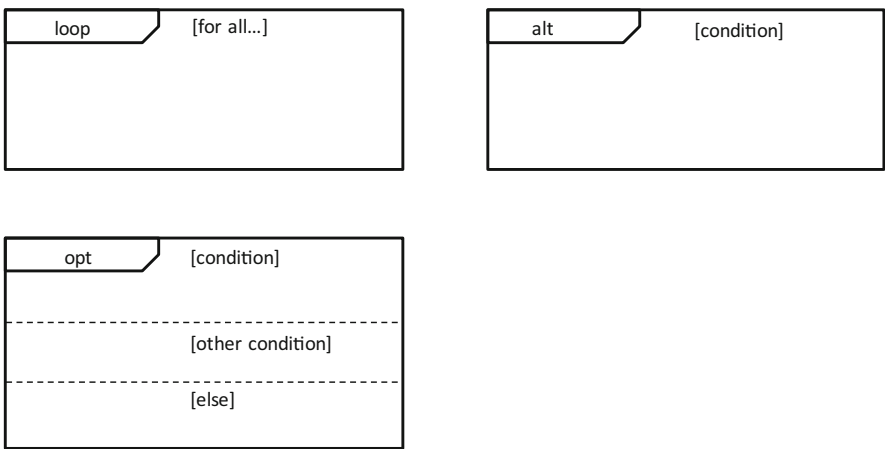

Fig. 13 Loops, alternatives and conditions used in sequence diagrams

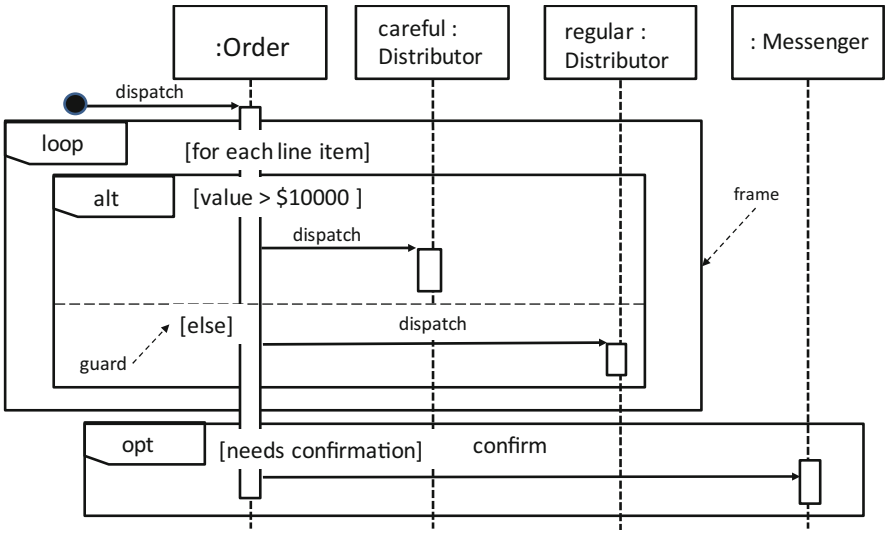

Fig. 14 Examples of loop, alternatives and options in a sequence diagram

Annex D: Analyses Required by IEC 61508:2010

Annex D gives an overview of the analyses required by annexes A and B in IEC 61508-3:2010. Note that not all of these analyses are required for all SILs. This annex is related to Fig. 6.1, Chap. 6. For more information on each analysis, see IEC 61508-7:2010. Some of the analyses are handled in more details elsewhere. This is indicated by the reference "See also. . .". *Note: references Ax and Bx below are to the standard.*

- A5—Software design and development
  - Dynamic analysis and testing—C.5.1: **Probabilistic testing. Aim:** To gain a quantitative figure about the reliability properties of the investigated software. Required for SIL 2, 3 and 4.
  - Data recording and analysis—C.5.2: **Aim:** To document all data, decisions and rationale in the software project to allow for easier verification, validation, assessment and maintenance. Required for all SILs.
- A8—Modification
  - Impact analysis—C.5.23: **Aim:** To determine the effect that a change or an enhancement to a software system will have to other software modules in that software system as well as to other systems. See also Sect. 8.2 in this book. Required for all SILs.
  - Data recording and analysis—C.5.2—see A5. Required for all SILs.
- A9—Software verification
  - Static analysis—B.6.4 and Table B.8. **Aim:** To avoid systematic faults that can lead to breakdowns in the system under test, either early or after many years of operation. Required for SIL 2, 3 and 4.
  - Dynamic analysis and testing—B.6.5 and Table B.2. **Aim:** To detect specification failures by inspecting the dynamic behaviour of a prototype at an advanced state of completion. Required for SIL 2, 3 and 4.
  - Offline numerical analysis—C.2.13: **Aim:** To ensure the accuracy of numerical calculations. Required for SIL 3 and 4.
- A10—Functional safety assessment
  - Failure analysis—see B.4. Required for SIL 3 and 4
  - Common cause failure analysis of diverse software (if diverse software is actually used)—C.6.3: **Aim:** To determine potential failures in multiple systems or multiple sub-systems which would undermine the benefits of redundancy, because of the appearance of the same failures in the multiple parts at the same time. Required for SIL 3 and 4.
- B2—Dynamic analysis and testing
  - Test case execution from boundary value analysis—C.5.4: **Aim:** To detect software errors occurring at parameter limits or boundaries. Required for SIL 2, 3 and 4.

- B4—Failure analysis—none of these are required for any SIL.
  - Event tree analysis—B.6.6.3: **Aim:** To model, in a diagrammatic form, the sequence of events that can develop in a system after an initiating event, and thereby indicate how serious consequences can occur. An event tree is difficult to build from scratch and using consequence diagram is helpful.
  - Fault tree analysis—B.6.6.5: **Aim:** To aid in the analysis of events, or combinations of events that will lead to a hazard or serious consequence and to perform the probability calculation of the top event. See also Annex B.12 in this book.
  - Software functional failure analysis—B.6.6.4: **Aim:** To rank the criticality of components, which could result in injury, damage or system degradation through single-point failures, in order to determine which components might need special attention and necessary control measures during design or operation. See also Annex B.8 in this book.
- B8—static analysis.
  - Boundary value analysis—C.5.4—see B2. Required for SIL 3 and 4.
  - Control flow analysis—C.5.9: **Aim:** To detect poor and potentially incorrect program structures.. Required for SIL 2, 3 and 4.
  - Data flow analysis—C.5.10: **Aim:** To detect poor and potentially incorrect program structures. Required for SIL 2, 3 and 4.
  - Static analysis of run time behaviour—B.2.2: Formal methods. **Aim:** Formal methods transfers the principles of mathematical reasoning to the specification and implementation of technical systems and therefore increase the completeness, consistency or correctness of a specification or implementation, and C.2.4. **Aim:** The development of software in a way that is based on mathematics. This includes formal design and formal coding techniques. Required for SIL 4.

References

1. *Commission Regulation (EU) No 347/2012 of 16 April 2012 implementing Regulation (EC) No 661/2009 of the European Parliament and of the Council with respect to type-approval requirements for certain categories of motor vehicles with regard to advanced emergency braking systems., in 347/2012*. 2012, European Commission: Belgium.
2. Burge, S. E. (2010). Systems engineering: Using systems thinking to design better aerospace systems. *Encyclopedia of Aerospace Engineering*.
3. Fowler, M. (2004). *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Professional.
4. IEEE. (1994). Std 1228 standard for software safety plans.

5. Myklebust, T., Stålhane, T., Hanssen, G., & Haugset, B. (2014). Change impact analysis as required by safety standards, what to do? In: *Probabilistic Safety Assessment & Management Conference* (PSAM12), Honolulu, USA.
6. Myklebust, T., Stålhane, T., Hanssen, G. K., Wien, T., & Haugset, B. (2014). Scrum, documentation and the IEC 61508-3:2010 software standard. In: *Proceedings of Probabilistic Safety Assessment & Management Conference (PSAM12)*. Oahu, USA: Self-Published.
7. Myklebust, T., Stålhane, T., Lyngby, N. (2016). *The Agile Safety Plan*. PSAM13.
8. Nuclear_Regulatory_Commission. (2011). Identification of failure modes in digital safety systems – Expert clinic findings, Part 2. In *Research information letter*.
9. Rajan, A., &Wahl, T. (2013). CESAR: *Cost-efficient methods and processes for safety-relevant embedded systems*. Springer.
10. Wullt, T. (2015). *Behavior under non-fault conditions*. Addalot.
11. Dobi, S., Gleirscher, M., Spichkova, M., & Struss, P. (2015). *Model-based hazard and impact analysis*. arXiv preprint arXiv:1512.02759.
12. Łukasiewicz, K. (2017). *Method of selecting programming practices for the safety critical software development projects – a case study*. Technical report no. 02/2017. Gdańsk University of Technology.
13. DOI Bureau of Land Management. (2010). *Aviation Risk Management Workbook*, April 2010.

Glossary

- ACK—Acknowledge
- AHL—Agile Hazard Log
- ATAM—Architectural Trade-off Analysis Method
- CIA—Change Impact Analysis
- CIAR—Change Impact Analysis Report
- CM—Configuration Management
- CR—Change Request
- E/E/PE—Electrical and/or Electronic and/or Programmable Electronic technology
- EUC—Equipment Under Control
- FAT—Factory Acceptance Test
- FFA—Functional Failure Analysis
- FIT—Failure In Time (1×10^{-9} failures per hour)
- FMEA—Failure Mode and Effect Analysis
- FMEDA—Failure Mode Effect and Detection Analysis
- FTA—Fault Tree Analysis
- HazId—Hazard Identification
- HazOp—Hazard and Operability studies
- HL—Hazard Log
- IF-FMEA—Input-Focused Failure Mode and Effect Analysis
- ISA—Independent Safety Assessor
- MTTF—Mean Time To Failure
- MTTR—Mean Time To Repair
- MVC—Model View Controller (a pattern)
- NRC—Nuclear Regulatory Commission
- PE—Programmable Electronic
- PFD—Probability of Failure on Demand
- PHA—Preliminary Hazard Analysis
- PoC—Proof of Compliance / Proof of Conformance
- QA—Quality Assurance
- RAMS—Reliability, Availability, Maintainability and Safety

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8

- RAMSS—Reliability, Availability, Maintainability, Safety and Security
- ROI—Return On Investment
- SAR—Safety Assessment Report
- SAT—Site Acceptance Test
- SFF—Safe Failure Fraction
- SIL—Safety Integrity Level
- SIS—Safety Instrumented System
- SRAC—Safety-Related Application Conditions
- SRS—System Requirement Specification
- SSRS—System Safety Requirements Specification
- TDD—Test-Driven Development
- TFS—Team Foundation Server
- THR—Tolerable Hazard Rate
- TÜV—Technischer Überwachungsverein
- UML—Unified Modelling Language
- VPN—Virtual Private Network
- V&V—Verification and Validation
- WP—Work Package

Index

G. K. Hanssen et al., *SafeScrum® – Agile Development of Safety-Critical Software*,
https://doi.org/10.1007/978-3-319-99334-8