FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

Agile Methods and Safety-critical Software

An Analysis Based on the Principles of Agility and Safety

Master Thesis

Christoph Schmiedinger

Submitted in fulfilment of the requirements for the degree of

Master of Science in Engineering (MSc)

University of Applied Science FH Campus Wien
Master Degree Program: Technical Management

Matriculation Number
1110536006

Supervisor:
DI (FH) Hans Tschürtz, MSc, MSc

June 8, 2013

“Intelligence is the ability to adapt to change.”

Stephen W. Hawking

Declaration:

“I confirm that this paper is entirely my own work. All sources and quotations have been fully acknowledged in the appropriate places with corresponding footnotes and citations. Quotations have been properly acknowledged and marked with appropriate punctuation. The works consulted are listed in the bibliography. This paper has not been submitted to another examination panel in the same or a similar form, and has not been published.“

Place, Date

Signature

Acknowledgement

I would like to thank all the people who supported and inspired me in writing my master thesis. I am thankful for their constructive criticism, their honest feedback and their engagement, which helped me to successfully achieve my high ambitions.

First of all I want to thank DI (FH) Hans Tschürtz, MSc, MSc for supervising the thesis. From the very first moment that I introduced my research questions to him, he was enthusiastic about this topic. That marked the beginning of an exceedingly successful collaboration that continued throughout the realisation of this thesis. No matter which questions arose, I always had an opportunity to ask as he was always open and helpful in his counsel. In addition I particularly want to thank Hans Tschürtz for advancing me in order for me to have the opportunity to present my work in front of a broader audience and to get in touch with other researchers.

Furthermore, I want to thank Dr. Andreas Gerstinger and Florian Loikasek, two colleagues from *Frequentis AG*. Both supported me throughout the time of my writing this thesis by giving feedback on a regular basis. My paper was greatly enriched by their knowledge of and experience in safety-critical and agile software development.

Last but not least I want to thank my family, my partner and my friends for their continuous support throughout my university studies. Thank-you for motivating me when I felt down and for your generous understanding during these stressful times of study and research.

Abstract

Increasing use of software, fierce competition and ever changing business demands lead to paradigm changes in the area of software development. Modern, lightweight and efficient software development approaches called agile methodologies have gained importance as they seem to address exactly those upcoming challenges. As these changes have come about also in the area of safety-critical software development, the question arises of whether the adoption of agile methodologies can leverage this particular kind of software development as well.

As there is limited experience and evidence of the adoption of agile methodologies in the area of safety-critical software development, these approaches are frequently confronted with prejudices. In order to dispel these objections, this thesis focuses on developing an agile procedure model that fits both safety and agility attitudes. For the purpose of considering both ideologies, an evaluation of their underlying principles is done to determine the most relevant synergies and conflicts. To ensure software safety, the model is developed in the context of EUROCAE ED-153, a guideline for software safety assurance in the air navigation service industry.

The result of this thesis is a holistic agile procedure model that allows software teams to be agile while ensuring safety by incorporating the necessary activities required by ED-153. Therefore the benefits ascribed to agile methods can be leveraged in order to develop valuable, high-quality and safety-assured software for future customer needs and demands.

Abbreviations and Acronyms

ACM	Association for Computing Machinery
ALM	Application Lifecycle Management
ANS	Air Navigation Service
ANSP	Air Navigation Service Provider
BPM	Business Process Management
CASCON	Centre for Advanced Studies on Collaborative Research
CESAR	Cost-efficient Methods and Processes for Safety Relevant Embedded Systems
CMMI®	Capability Maturity Model® Integration
COTS	Commercial Off The Shelf
CSR	Corporate Social Responsibility
CTIT	Centre for Telematics and Information Technology
DASC	Digital Avionics Systems Conference
DoD	Definition of Done
DoD	Department of Defence
DRDO	Defence Research & Development Organisation
DSRoE	Derived Safety Requirements on Elements
EC	European Commission
EMOSIA	European Model for ATM Strategic Investment Analysis
ESARR	EUROCONTROL Safety Regulatory Requirements
ESEM	Empirical Software Engineering and Measurement
EUROCAE	European Organisation for Civil Aviation Equipment
EUROCONTROL	European Organisation for the Safety of Air Navigation
FDD	Feature Driven Development
FHE	Functional Hazard Evaluation
FMEA	Failure Mode and Effective Analysis
FOSE	Future of Software Engineering
FTA	Fault Tree Analysis
GSN	Goal Structuring Notation
HAZOP	Hazard and Operability Study
HMI	Human-machine Interface
ICAO	International Civil Aviation Organization
ICB	IPMA Competence Baseline
ICIS	International Conference on Information
ICSE	International Conference on Software Engineering

ICSEA	International Conference on Software Engineering Advances
IEC	International Electrotechnical Commission
IFIP	International Federation for Information Processing
IJCSI	International Journal of Computer Science Issues
IPMA	International Project Management Association
ISaPro®	Integrated Safety Process
ISO	International Organization for Standardization
ISSC	International System Safety Conference
ISW	Information Survivability Workshop
IT	Information Technology
ITEA	Information Technology for European Advancement
MIT	Massachusetts Institute of Technology
NASA	National Aeronautics and Space Administration
OFR	Office of the Federal Register
OOPSLA	Object-oriented Programming, Systems, Languages & Applications
OSSE	Operational System Safety Evaluation
PFH	Probability of Dangerous Failure per Hour
PHA	Preliminary Hazard Analysis
PHI	Preliminary Hazard Identification
PP	Pair Programming
PSP	Personal Software Process
PSSE	Preliminary System Safety Evaluation
ROI	Return on Investment
RSSB	Rail Safety and Standards Board
RTCA	Radio Technical Commission for Aeronautics
SAAPM	Safety Assured Agile Procedure Model
SAE	Society of Automotive Engineer
SCFL	Safety Critical Function List
SEI	Software Engineering Institute
SIL	Safety Integrity Level
SPI²	System & Software Process Improvement and Innovation
SPICE	Software Process Improvement and Capability Determination
SSDA	Software Safety Design Analysis
SSE	System Safety Evaluation
SSRA	Software Safety Requirements Analysis
SWAL	Software Assurance Level
TDD	Test-driven Development
TSP	Team Software Process

VISSE	Vienna Institute for Safety & Systems Engineering
XP	Extreme Programming

Table of Contents

1 Introduction and Purpose

1.1 Background

Increasing globalisation in combination with tougher competition is an evolution which has been observable over the last few decades in every technology branch. Companies and enterprises which have successfully positioned themselves as high-quality software producers in previous years are nowadays confronted with shorter innovation cycles and the continuously changing demands of the market [Mac03, VB09]. These effects are further intensified by the increasing complexity of systems and software within the context of information technology [GH12].

These factors have found their way into the domain of safety-critical applications as well [Hei07]. Although quality and safety are of the utmost importance in this domain, the market and customer needs have become more crucial. In terms of quality and safety, these software systems have to comply with certain general and domain-specific safety standard specifications. While customers consider safety compliance, quality and reliability as a matter of course, the focus on functional requirements has been increasing continuously [Gar09].

In order to overcome these new challenges and satisfy corresponding requirements, new approaches within the context of software development were introduced in the 1990s. One of these emerging modern approaches is agile software development. This term groups various approaches whose aim is to provide a lightweight, efficient and more flexible development process compared to traditional approaches [NMM05]. The benefits ascribed to agile methods are attractive to software development teams working in the safety-critical software domain as well. This attractiveness is mainly caused by the traditional way, in which safety-critical systems are developed according to a rigorous heavyweight process that emphasises an upfront design and the production of documentation [GPM10]. Safety standards highly influence this process by mandating steps to be followed or evidence to be delivered from the process. Such procedure models are wide spread in safety-critical development due to their wide acceptance, their thorough definition and the fact that they have been the best practices for many years [GPM10].

Agile procedure models are challenged mainly because of their lightweight approach and their attitude to processes and documentation, which seem to be inappropriate for the proof of safety. Some of these perceptions are caused by prejudices about agile approaches in projects such as the lack of discipline and/or documentation [Hol06]. In addition to this uncertainty, there is limited industrial experience and evidence of how to successfully adopt these practices in the domain of safety criticality. Those are the main reasons why industry is still so cautious when it comes to the adoption of these modern software practices [LBB+02].

One business area, in which these previously indicated trends can actually be observed is the field of software development for air traffic management. A draft plan for global air navigation capacity and efficiency, published by the International Civil Aviation Organisation (ICAO), outlines the fact of extensive software use, particularly in the area of air navigation services [ICAO12]. In this plan, which focuses on the years until 2028, the ICAO mentions the terms “cloud applications” and “software as a service” [ICAO12]. These references strongly indicate that software is one of the key enablers for meeting the future strategic objectives in air navigation.

In such safety-critical areas, standard specifications for the development of hardware and software are common today. Due to the fact that the use of software is still increasing, it is becoming necessary to develop guidelines specifically focused on software development. This is especially important because of the huge multiplicity of approaches and programming languages which are used nowadays [HSV+12, Kin11]. In August 2009, the European Organisation for Civil Aviation Equipment (EUROCAE) issued the guidance ED-153, which particularly applies to the software parts of systems within air navigation services [EUROCAE09]. ED-153 and its related standard specifications in other industry segments only mark the beginning of an engagement in safety assurance within software systems and applications. In combination with the various development approaches available, these aspects will be some of the main topics that safety has to deal with in the near future [LCF13].

1.2 Research Objectives

As pointed out in the introductory chapter, there is a lot of uncertainty about adopting agile methods within the area of safety-critical software development. Furthermore, research about the suitability and applicability of agile methods in such industries is still at an early stage [GPM10] as successful adoptions and experiences reported in literature are very rare.

These concerns lead to the following central research question:

> Can agile methodologies be used to develop safety-critical software applications?

By raising this research question, the following hypothesis can be proposed:

> It is possible to use agile methodologies successfully in the development of safety-critical software, if that usage is diligent and thoughtful. To achieve this, an appropriate procedure model including suitable methods must be used.

The scope of this thesis is to investigate why agile methodologies conflict with safety-related issues and whether these conflicts can be overcome by using an adapted agile procedure model for safety-critical software development. Within such a model, it is necessary to consider the tasks and activities required in order to ensure safety. Therefore the whole investigation is accomplished in the context of EUROCAE ED-153 [EUROCAE09], a guideline on software safety assurance. The results of this investigation

should support the decision-making of organisations that deal with the issue of adopting agile methodologies within the safety-critical area.

1.3 Research Method

In order to answer the research question and prove the proposed hypothesis, a defined research method is necessary. Figure 1 depicts the four steps of the research method that is applied in this thesis.

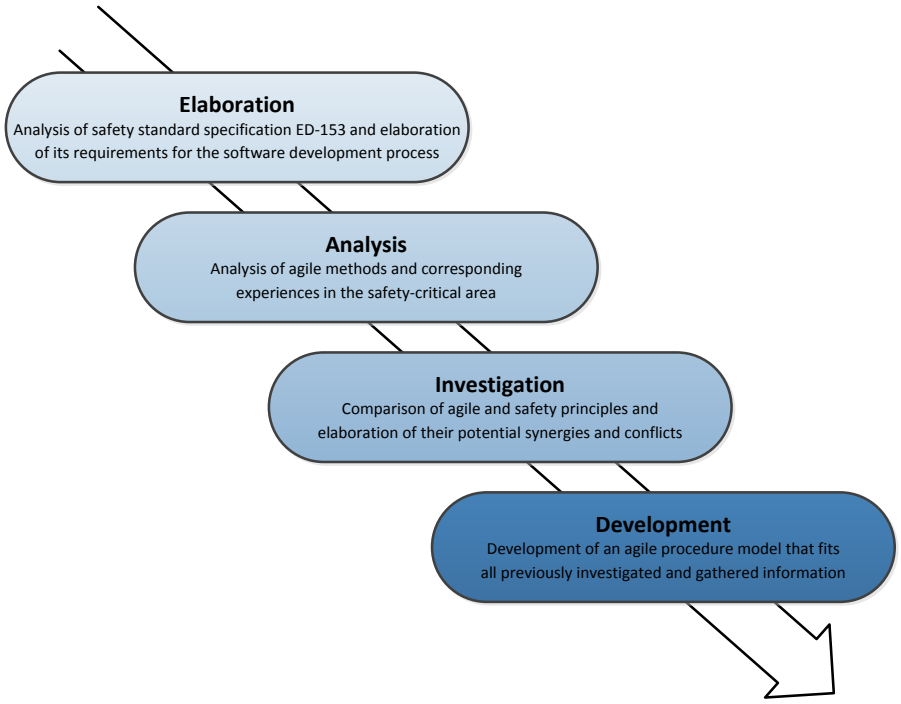

Figure 1: Research Method

The first phase of the research method consists of the analysis of the EUROCAE ED-153 [EUROCAE09] guideline for software safety assurance and the elaboration of its requirements regarding the software development process. In the analysis, all applicable objectives are mapped to processes of a generic development procedure model called ISaPro® [TKH12] that is specifically tailored to the safety-critical industry. This mapping should help to identify all necessary activities in order to be compliant.

The analysis of agile methods including their values, principles and approaches is the focus of the second phase. Furthermore this chapter deals with the technical practices that are commonly used in conjunction with agile methods. An overview of the published results of scientific researches and the interdependencies between agile and traditional approaches complete this research phase.

The third phase of the research method comprises the evaluation of the principles of safety and agility and shows how these principles interact. In order to identify their potential synergies and conflicts, the attitudes, processes and technical practices of both approaches are examined.

The development of an agile procedure model that fits into safety-critical software development is the central topic in the fourth and last phase of the research method. This model is created considering the principles of safety and agility. In addition, this research phase is influenced by the requirements that are imposed on the software development process by ED-153.

1.4 Related Work

As outlined in the Background (see chapter 1.1), agile methodologies have advantages in environments that are challenged by continuously changing demands and requirements. To leverage these benefits, the topic of adopting agile methodologies in the area of safety-critical development is of great interest. This is also supported by numerous scientific papers that were published in the last few years. The most relevant articles including their results are described in this section.

An Iterative Approach for Development of Safety-Critical Software and Safety Arguments written by Ge, Paige and McDermid [GPM10], is a conference paper that deals with aspects of this thesis. Their paper addresses the notion of up-front design and the key difficulties that appear when developing safety-critical software iteratively.

Their first result was the generation of a generic software development lifecycle model for agile methods in order to reduce different agile methods to a common denominator. This lifecycle consists of four phases: preparation, planning, short iterations to release, and integration. When the generic agile model was compared to traditional approaches, the authors identified two major differences within the traditional approaches: the use of an up-front design and a very monolithic way of implementation. [GPM10]

Regarding the first issue, the up-front design, the authors agree with the agile methodology of creating the shape of the system first, followed by developing its detailed design iteratively during implementation. Nevertheless this shape must be detailed enough to provide sufficient input for the hazard analysis (also recommended in other papers and studies by Garg [Gar09] and Bozheva et al. [BHI+05]). In order to achieve this sufficiency, the system architecture model and the main functional requirements for each component have to be produced in this initial phase. [GPM10]

The second issue raised by the paper is the iterative development of safety-critical software. The main concerns referred to the plan to produce the safety argument of the system iteratively. In order to answer these concerns, the authors recommend the

construction of modular safety arguments, e.g. for each software module. The system safety argument finally consists of all modular component safety arguments and one argument that deals with the interactions of those single modules. [GPM10]

Their conclusion is that agile practices may not change the nature of the entire safety-critical development procedure model, but can improve the agility of this development bit by bit [GPM10].

In another paper entitled *Agility and Lean for Avionics* the author Chenu [Che09] describes how agile approaches have brought value to avionics. In the paper he points out that in his opinion safety analyses are not feasible to assess the amount and different kinds of software errors. Therefore organisations developing safety-critical software should impose rigour on the development process in order to be confident that safety is ensured.

According to Chenu [Che09] agile practices, particularly test-driven development (see chapter 5.2.2), contribute to more efficient development and bring value to the certification of the software. The author even stated that, in his opinion, automatic and repeatable tests have a great advantage over manual ones. A conclusion he reached having witnessed distracted and bored employees testing safety-critical software on several projects in which he was involved. [Che09]

Based on Chenu's experience, Extreme Programming (see chapter 5.3.1) as an agile methodology can be used for software that has to be certified. The main problems during the certification process are caused by requirements and traceability. That is why these issues have to be considered very carefully when adopting an agile method. In Chenu's opinion, agile values and principles (see chapter 5.1) fit with safety-critical software development apart from the stance on documentation. As documentation is essential for certification, it has to be written and developed iteratively. [Che09]

Another hypothesis proposed by Chenu [Che09] is that *"[…] it is easier to make a correct program fast than it is to make a fast program correct"*. Therefore he recommends preventing premature code optimisation. Instead, performance should be continuously monitored and improved based on these hard facts. Another important consideration when adopting agile practices is the determination of the iteration length. Chenu [Che09] recommends using longer intervals, e.g. four weeks, in safety-critical developments, due to the high complexity that such projects usually have to deal with.

Chenu's [Che09] conclusion is that agile and lean practices help to grow high-integrity products, while reducing costs. Both organisational and engineering practices have to be combined in an effective way while imposing rigour and strict discipline on them. Technical excellence has to be the target of utmost importance in order to succeed.

A related article – older but still valid – written by Poppendieck and Morsicato [PM02] in 2002 is *XP in a Safety-critical Environment*. Like Chenu's article, it discusses the experience of using Extreme Programming for developing software that has to be certified. The authors noticed a conflict between the identification of all hazardous conditions in the beginning and the safe-guarding that all upcoming changes do not influence existing hazard controls. [PM02]

Morsicato [PM02] proposed the hypothesis that *"[…] it is dangerous to think that all the safety issues will be exposed during an initial design"*. According to his opinion, it is far better to re-evaluate safety issues on a regular basis, based on what has been educed from development (this is also recommended in another study by Vuori [Vuo11]). This continuous task should be facilitated by a parallel refactoring of the software in order to achieve a simple design. This simplicity would subsequently lead to a safer design, allowing the developers to concentrate on safety. [PM02]

A further important point of consideration is the unit test framework of the software. The authors recommend an emphasis on rigorous in-line testing with the help of unit tests (which is in line with statements provided by Bozheva et al. [BHI+05]). The advantage of this approach is that in the case of problems appearing, the developers just have to take a look at the interfaces for the causes of the defects. [PM02]

The conclusion of the article was that the examined project did not pass the audit by the customer in the end. However, this was not because of the fact that they were using Extreme Programming. It was simply the fact that the organisation had no defined process for such an agile development method. Therefore the auditor objected to the fact that the development team was allowed by the organisation to implement new methodologies unilaterally. Based on their experiences, both authors agreed that it is definitely possible to implement agile approaches in the industrial area of safety-critical systems. [PM02]

Many scientific articles such as these three indicate that the adoption of agile methodologies within the development of safety-critical software is possible. In most cases this is done either by describing how to adopt single agile values or principles in order to fit safety-critical developments or by conducting a survey in the software development industry. Only a minority of the academic articles that have been published had the research goal of developing a kind of agile procedure model that is designed for applying agility and safety. Given that, this thesis proposes the development of a holistic agile procedure model based on the objectives of EUROCAE ED-153 and the values of agility in order to generate scientific findings for the adoption of agile methods.

1.5 Overview

This thesis is structured as follows:

Chapter 2 introduces the basic safety aspects when dealing with software development. It aims to give an overview of the various terms used, the characteristics of software safety, the safety standard families and some of the methods and techniques used during the analyses.

Chapter 3 deals with the introduction to the EUROCAE ED-153 [EUROCAE09] guidance on software safety assurance. First it describes its definition of the purpose, scope and perception of a software safety assurance system. In addition it points out all the applicable processes and objectives of ED-153 within the scope of this thesis.

Chapter 4 presents a generic procedure model which is especially tailored to safety-critical development: the ISaPro® [TKH12]. It facilitates the determination of activities that are necessary to satisfy the relevant objectives of the EUROCAE ED-153 guidance.

Chapter 5 gives an overview of the agile methodologies. This includes their values, principles and the technical practices that are recommended by them. Additionally, this chapter refers to relevant scientific studies and the interdependencies between agile and traditional software development methods.

Chapter 6 explains the model of the objective mapping process from the ED-153 guidance to the ISaPro® framework. In addition it presents the condensed results of this process in a tabular formatted overview.

Chapter 7 outlines the evaluation of the principles of safety and agility. Furthermore this chapter deals with the identification of potential synergies and conflicts between the two approaches.

Chapter 8 is the core part of this thesis. It describes the agile procedural model which has been developed for organisations adopting agile methodologies in the area of safety-critical development. The chapter comprises a detailed description of the model, its compliance with ED-153 and its evaluation in relation to certain topics.

Chapter 9 concludes the thesis with a summary of the results.

Annex A comprises the detailed results of the objective mapping process, where each objective of ED-153 is mapped to one ISaPro® process. It also includes a summary of ISaPro® processes showing which of these processes fulfils which ED-153 objectives.

Annex B consists of the adapted integrated procedure model that has been tailored and extended in order to be compliant with EUROCAE ED-153. This model is described by a comprehensive list of activities for each ISaPro® process. In addition, these activities are linked to the different phases of the agile procedure model (developed in chapter 8) in which they should be conducted.

2 Safety Aspects in Software Development

Due to the focus of this thesis on safety assurance within the development of software, this chapter will deal with the basics of safety and in particular the safety aspects in software development.

2.1 Terminology

Apart from the term >safety-critical systems<, a few basic terms should be defined in order to achieve a common point of view. Many different definitions of the various terms used in this paper can be found in scientific literature, therefore only the most suitable of these sources will be cited.

2.1.1 Safety

The term >safety< is defined in many different ways in literature and standard specifications. For the purpose of this thesis the definition by the British Rail Safety and Standards Board (RSSB) is a very appropriate one. It points out that it is not only users who are affected by the systems' or products' safety; the whole general public might be affected too and therefore such safety issues should be avoided.

> *"The avoidance of death, injury or poor health to customers, employees, contractors and the general public, caused by occupational accidents, incidents or hazards, also avoidance of damage to property and the environment."*
>
> *Rail Safety and Standards Board [RSSB93]*

According to Avižienis et al. [ALR+04], safety is embedded in the holistic concept of dependability. To be more precise, safety is, along with availability, reliability, integrity and maintainability, an attribute of dependability as depicted in Figure 2. Dependability is defined as the ability to deliver a service that can justifiably be trusted. Safety is defined as the absence of catastrophic consequences affecting the user(s) and environment in this context. [ALR+04]

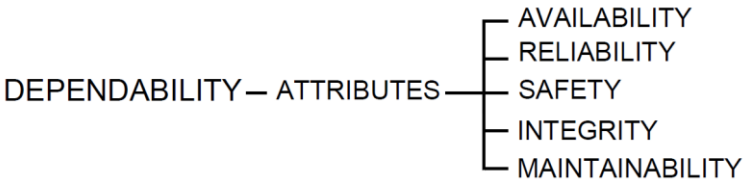

Figure 2: Dependability Attributes [based on ALR+04]

2.1.2 System Safety

System safety is defined in many military standards (e.g. DoD MIL-STD-882E [DoD12]) and also by the US Air Force Safety Agency as follows:

> *"The application of engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle."*
>
> *Air Force Safety Agency [AFSA00]*

System safety is considered as a term with quite a vast range of meanings. To accurately define system safety it is necessary to decide whether the system consists of only one simple element or numerous subsystems which presumably have various dependencies on each other [Wel02]. This thesis will primarily focus on complex software systems with various subsystems and therefore the term >software safety< – defined in chapter 2.2 – is more appropriate.

The activities covered by system safety focus on identifying, analysing and assessing hazards in order to set preventive measures to avoid hazardous situations (see chapter 2.1.3) [NASA04].

2.1.3 Hazards

According to Leveson [Lev11], a hazard is defined as a system state or set of conditions, which, together with a particular set of worst-case environmental conditions, will lead to an accident or loss. Some definitions use a set of events rather than a set of conditions, but both can be used if they are used consistently. Another definition provided by EUROCAE [EUROCAE09] is that a hazard is a potential risk situation in which one or more causes lead to one or more consequences that are a potential source of harm (see Figure 3).

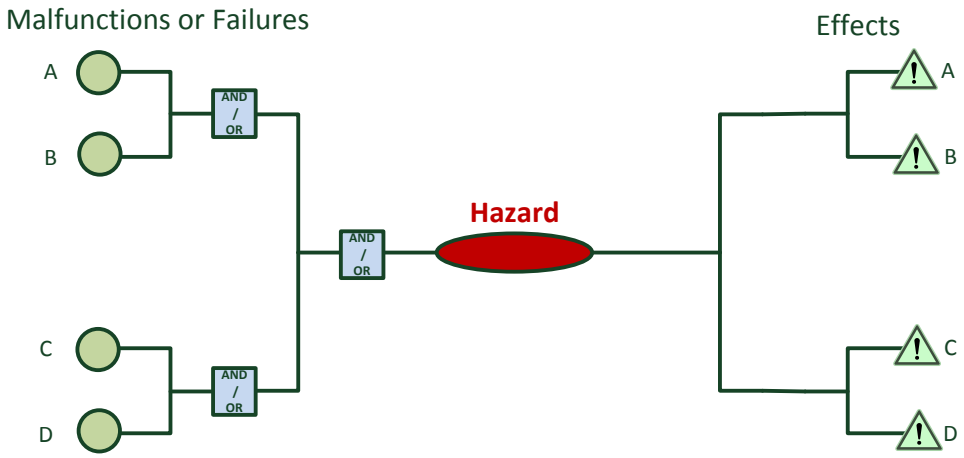

Figure 3: Relationship between Malfunctions or Failures, Hazards and Effects [based on EUROCAE09]

Regardless of which of the definitions is chosen, hazards are basically kinds of preconditions that occur on the boundaries of a system and can lead to an incident or

accident. To ensure the detection of hazards, it is necessary to define the system as accurately as possible to investigate its boundaries for possible hazards. System safety is responsible for implementing controls for any of the identified hazards that cannot be accepted as tolerable risk. These controls reduce either the likelihood of the cause or the impact of the consequence or both [EUROCAE09].

2.1.4 Safety-critical Systems

A term that has to be distinguished from system safety (see chapter 2.1.2) is >safety-critical system<. This is synonymous with the term or >safety-relevant system<. Both terms are widely used and their distinction has become blurred. Safety-critical systems tend to be those systems in which a single failure leads to a fatality or strongly increases the risk to the environment [SS04]. Systems in which a single failure is not necessarily critical, and another coincident failure of some other item must occur for there to be a fatality, tend to be called safety-relevant systems [SS04].

The term >system safety< – defined in chapter 2.1.2 – is the term for a process-oriented view of safety aspects. Safety-critical systems are in fact those systems which could cause harm to humans, property or the environment. According to Knight [Kni02], the term could be considered within a broader scope:

> *"If the failure of a system could lead to consequences that are determined to be unacceptable then the system is safety-critical."*
>
> *John C. Knight [Kni02]*

Along with the term >safety-critical systems<, there are specific industries or domains which are generally considered as safety-relevant. This is based on the fact that there is a high probability that system failures will cause harm to humans, property or environment. These domains are:

- Aerospace and aviation
- Automotive industry
- Pharmaceutical industry
- Automation
- Defence
- Infrastructure

2.1.5 Safety Integrity Level (SIL)

The safety integrity levels were introduced in various specification standards (including IEC 61508 [IEC10]). They require that a certain safety level has to be assigned to processes which have insufficient mitigation from potential hazards. In order to minimise their potential impact it is necessary to add safety functions or systems to these processes. This should ensure functional safety. In the IEC 61508 specification there are four SILs, where SIL 1 is the lowest and SIL 4 the highest level of safety integrity [IEC10]. SILs are basically a measure of the reliability of the safety-related system regarding its avoidance of dangerous failures. So the assignment of the level is based on the required

availability of a safety-related function. The higher the risk of the system, the higher the required availability has to be and depending on that the higher the allocated SIL is. [Hya03]

Table 1 shows the failure rates for dangerous failures per safety integrity level according to IEC 61508 [IEC10]:

Safety Integrity Level (SIL)	Average frequency of a dangerous failure of the safety function (PFH) [h^{-1}]
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 1: Safety Integrity Levels (SIL) [IEC10]

Due to the focus on failure rate, the SIL is more appropriate on hardware than on software (see the quote by Nancy Leveson in chapter 2.2). Therefore different standards use different classifications for categorising systems into classes of safety-criticalness. Due to the focus of this thesis on software and EUROCAE ED-153 [EUROCAE09], their level, called the Software Assurance Level (SWAL), will be introduced in chapter 3.2.

2.1.6 Threats: Faults, Errors and Failures

According to Avižienis et al. [ALR00], threats are factors endangering dependability (see chapter 2.1.1). Figure 4 shows the division of these threats into three categories.

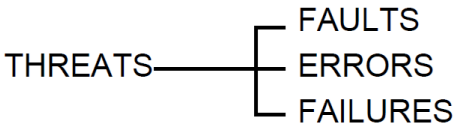

Figure 4: Categories of Threats [ALR00]

When a service implements a specified system function, a correct service is delivered. In the case that the delivered service deviates from the correct service it is called a system failure. There are various reasons why a system may fail: probably the system does not comply with the specification or the specification does not accurately describe the function. More precisely, the failure is the transition between a correct and an incorrect service. The timespan until the service is restored is called outage. [ALR00]

The trigger for such a failure is an error which reaches the service interface. A service failure therefore means that at least one of the external states of the system deviates from the correct service. The cause for this deviation is called the error. It is important to

mention that many of the errors do not reach the service interface and therefore remain unnoticed. [ALR00, ALR+04]

A fault is the suspected cause of an error. It might be the case that many of the faults are dormant. When the fault actually leads to an error, it is considered as an active fault. [ALR00, ALR+04]

As the previous three paragraphs indicate, faults are preconditions for errors and errors are preconditions for failures. Figure 5 depicts this fact, which is, according to Avižienis et al. [ALR00], also called the *"fundamental chain of threats"*. Failures are always visible at the system boundary, whereas faults and errors cannot be perceived there. The last transition after the failure has occurred can lead to different situations. Failures may cause new or dormant faults or even potential sources of harms, e.g. hazards (see also Figure 3). [ALR00]

Figure 5: Fundamental Chain of Threats [ALR00]

2.1.7 Safety Management

Safety management is a business-like approach to safety; according to Schedl et al. [SW08] it is defined as follows:

> *"Safety management is a pro-active and reactive discipline aiming at minimising the risk of an accident as far as reasonable practicable."*
>
> *Gabriele Schedl et al. [SW08]*

According to this quote the philosophy of this approach focuses on prevention. Another important fact is that the responsibility for systematic safety management has to start at the very top of the organisation and cascades down the hierarchy. Safety managers are the main driving force within the context of the company for establishing and co-ordinating an effective strategy for safety management. These employees have to ensure that the scope of safety management is companywide. As with the majority of management systems and processes, there is a strong focus on the continuous improvement of safety management as well. [SW08]

Safety management consists of the following key aspects:

- Documents (e.g. a safety policy or safety handbook) [Lev11]
- Competence and independence of safety engineering employees [IEC10]
- Safety lifecycle including safety activities using well-known methods (see chapters 2.4 and 4.3) [IEC10]

2.2 Software Safety

Due to the fact that this thesis deals with safety-critical software, the term >software safety< is more appropriate. In general this term customises the term >system safety< (see chapter 2.1.2) to software.

One of the major differences between software and system safety is that software cannot cause harm directly to humans or the environment. However software is typically used for operating an electronic system (e.g. a computer) or controlling other hardware parts; therefore it can either lead directly to a hazard or it can be used to control hazards. This kind of software is called hazardous software. [NASA04]

> *"Software does not fail – it just does not perform as intended."*
>
> *Nancy Leveson [NASA04]*

Safety-critical software includes the previously mentioned hazardous software and all kinds of software which influence it. According to the NASA Software Safety Guidebook [NASA04] the term covers the following types of software:

Software that …

- … controls or monitors hazardous or safety-critical hardware or software
- … provides information which is necessary for safety-related decisions
- … performs off-line processes or is used for analysis of safety-critical software (e.g. software for verification of hazard controls, modelling and simulation programs used for simulating the operational behaviour of a safety-critical system)
- … resides on the same physical platform with safety-critical software

2.2.1 Relevance

This section outlines the practical relevance of software safety in today's systems. First of all, the general use of software in our lives has been increasing continuously. This is mainly caused by the attempt to leverage software for all the actions in daily life that can be automated. In combination with the aims of reducing costs and gaining performance, this is leading to a steady trend toward higher complexity [BV10]. The fact of increasing interoperability between systems is further reinforcing that trend [Wal04].

In 2009 a report by NASA determined that the size of flight software in space shuttles is growing exponentially over time (1969 – 2005) [Dvo09]. According to Bozzano et al. [BV10] similar trends can be observed in other domains such as avionics, automotive and switching systems. Along with software size, software complexity is increasing too. This can be seen from the increasing number of functions and states and the discontinuous behaviour of software itself, where a little variation in one program input could cause a great variation in one output [BV10].

Apart from software size and complexity, there is a social aspect leveraging the trend towards considering safety in software systems as well. As many examples show, the risk acceptability in society is continuously decreasing. This can be associated with the majority of customer or user needs in accordance with the Kano model, where the innovations of yesterday are the basic needs of tomorrow [KST+84]. The passenger airbag in a car is one example of a safety function which is basic equipment nowadays but in the early 1990s was only part of upscale configurations.

These trends indicate that there is a need to engineer complex cross-linked safe software systems in order to meet the high expectations and requirements of today's society.

2.2.2 Challenges

Reliability and availability are central topics of system safety (see chapter 2.1.2). The IEC for example defines the safety integrity levels (SILs) (see chapter 2.1.5) as values for reliability with respect to dangerous failures [IEC10]. Reliability is in turn coupled with availability under the premise of constant maintainability [MTL10].

Reliability is especially important when it comes to hardware such as mechanical or electronic components. These components have an average life span under certain conditions which is influenced by environmental and operational impacts. By means of statistics mean failure rates can be calculated. In contrast, the reliability of software is hard to determine. As the quote by Leveson (see chapter 2.2) indicates, software does not fail, break down or wear out. But software has a large number of states in comparison to hardware. Thus it is not economical or even possible in larger software projects to test all those states. These facts lead to the conclusion that all software failures are caused by systematic faults in development or operation. Therefore IEC recommends qualitative techniques and evaluations. [IEC10, NASA04]

While reliability of software cannot be measured exactly or tested exhaustively, NASA [NASA04] recommends that the following system characteristics be determined for estimating the effort that is required to meet the targeted safety level:

- Degree of control over safety-critical functions
- Software system complexity
- Timing criticality of control actions

2.3 Safety Standards

Since safety is such a sensitive issue, there are various safety standards, regulations and various types of guidance in place. Typically safety-critical systems require certification or assessment based on specific standards in order to permit the transition into operation. These standards define a number of accepted ways of developing safe systems.

In the course of the development of safety-critical systems there is the challenge of having dozens of different safety standards to fulfil. While some of them are generic approaches, others concentrate on specific domains or industries. Figure 6 provides an overview of safety standards grouped by their application. While IEC 61508 [IEC10] is a generic standard, many other industry-standards have been derived from it, such as the ISO 26262 for the automotive industry. In particular industries, such as avionics or in the military industry, even custom standard specifications are available, which in fact have some further derived and related standards.

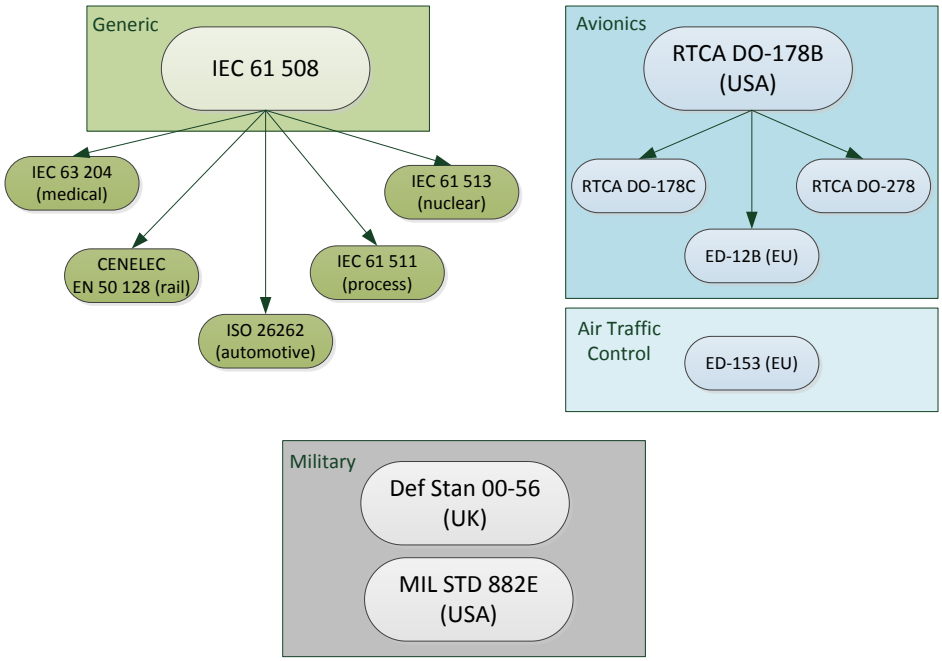

Figure 6: Safety Standard Families [based on Gar12, SW08]

Common topics within safety standards are:

- Description of development approach including all relevant activities
- Detailed description of safety process or procedure model
- Scope of risk and hazard identification techniques
- List of necessary formal safety analysis throughout the whole lifecycle
- List of required documents for approval

2.4 Safety Analysis Methods & Techniques

In order to ensure system or software safety it is common to use generally accepted safety analysis methods and techniques. This chapter gives a brief overview of the most reasonable ones. It has to be pointed out that during the safety lifecycle (see chapter 4.3) it is necessary to employ more than one specific method. An example of the combination

of different methods and techniques is the usage of an inductive (e.g. FMEA, see chapter 2.4.2) and a deductive (e.g. FTA, see chapter 2.4.4) method.

2.4.1 Preliminary Hazard Analysis (PHA)

The preliminary hazard analysis (PHA) uses the preliminary hazard list (which is initially created and based on the technical concept of the system) as its input and further expands and develops it. The first task is to identify general hazard groups in order to simplify, cluster and categorise the preliminary hazard list. The PHA is one of the most critical analyses because of its first attempt to isolate the hazards of a system. It will provide reasonable hazard controls and indications where further analyses are needed due to the criticality of the system's part. [Hya03, Vin06]

2.4.2 Failure Mode and Effective Analysis (FMEA)

The Failure Mode and Effective Analysis (FMEA) is an inductive bottom-up approach used to determine the reliability of a system. It is designed for evaluating a system or a subsystem to identify all possible failures of each individual component including a forecast of their effects on the analysed level and the next higher level. This is done by assessing all possible hazards by determining their likelihoods and severity. Furthermore this list of possible failures is augmented by recommendations for mitigating the identified hazards in order to reduce or even remove them. FMEA supports the safety engineering process on different levels during the whole lifecycle, although the analysis is commonly used very early in the system development on the component level. [BV10, Vin06]

2.4.3 Hazard and Operability Study (HAZOP)

According to Vincoli [Vin06] the definition of the hazard and operability study (HAZOP) is a *"systematic investigative study, which has the goal to examine potential deviations of operations that could result in problems or hazards*". This method is particularly appropriate for analysing the system's interfaces. Critical success factors of this method are on the one hand the experience and expertise of the attendees and on the other hand the communication process between them. The objectives of the study are to predict accidents by using information from previous analyses (e.g. the preliminary hazard analysis) and to discuss them in order to identify specific safety aspects and requirements. In addition it is of importance that the necessary reference data is available for supporting the analysis. This approach should result in the determination of appropriate design considerations for the purpose of accident prevention. [RCC99, Vin06]

2.4.4 Fault Tree Analysis (FTA)

The fault tree analysis (FTA) is a deductive method of logic which is especially used for very complex or detailed systems. In contrast to FMEA (see chapter 2.4.2), this method is a top-down approach, whereby the logic moves from the general to the specific level. Therefore it is used for examining possible conditions that lead to an undesirable event. This event is considered as the general or known outcome of a possible series of events and is the top event in this analysis. The aim of this analysis is the identification of

specific events that contribute to the top event, which results in the construction of a tree: the fault tree. The contributing factors can be clustered by their origin in order to allow accurate identification of where breakdowns can occur, if and what relationships exist, and which interfaces are affected. [Eri05, Vin06]

2.5 Safety Case

The safety case is also often referred to as the safety justification or safety assessment report [Sto96]; Wilson et al. [WKM97] define its purpose as follows:

> *"The purpose of a safety case is to present a clear, comprehensive and defensible argument supported by calculation and procedure that a system or installation will be acceptably safe throughout its life (and decommissioning)."*
>
> *Wilson et al. [WKM97]*

In order to fulfil safety certification standards, it is necessary to provide structured arguments and supporting evidence that the risks associated with the system have been considered carefully and appropriate actions have been taken in order to minimise them. The safety case therefore contains the description of the design and assessment methods used in the development process of a system. As this document is designed for third parties as well, it has to be as precise and clear as possible. This should help to support external parties such as certification or public authorities in confirming the safety of a product or system. [Sto96, TKH12, WKM97]

The representation of such a safety case can be either textual or in a graphical notation. While in most cases the textual ones are single linear documents that link results contained in other deliverable documents [WKM97], the graphically notated ones mostly use the so-called "Goal Structuring Notation" (GSN), developed by Kelly [Kel98]. This graphical technique is used to explicitly document the elements of any argument and the relationships between them. The main purpose is to demonstrate how claims concerning the safety of a system are divided into sub-claims until it can be supported by a body of evidence, e.g. the documented results of a safety analysis [TKH12]. Further advantages of GSN are its reusable patterns, reduced fault probability and the standardised framework [KW04, York11], which are further reasons why this method is widely used in industry [KW04].

3 EUROCAE ED-153 Guidance

While chapter 2.3 provides a brief overview of the safety standard families, this chapter is intended to introduce EUROCAE ED-153 [EUROCAE09]. EUROCAE ED-153 is a guideline for software safety assurance specifically in the area of air navigation service (ANS). Henceforth this guideline will be used as the basic input for the development of the agile procedure model (see chapter 8).

3.1 Purpose and Scope

Today a rising percentage of safety-critical air navigation service functions rely on automated processes, which are supported by software in many cases [EUROCAE09, Zem08]. This fact gives rise to new challenges for ensuring the required level of safety for this set of functions. The European Organisation for Civil Aviation Equipment (EUROCAE), a non-profit organisation which deals with the standardisation of electronic equipment in aviation, has therefore published the ED-153 [EUROCAE09] guidance. This guideline gives information on how to assure that the risk associated with deploying software within air navigation services is reduced to an acceptable level.

Content of the ED-153 guideline [EUROCAE09]:

- Recommendations and requirements for providing software safety assurance
  - Per major process in the software lifecycle
  - Per software assurance level (SWAL) (see chapter 3.2)
- References to other standards dealing with safety assurance (e.g. IEC 61508 [IEC10])
- Guidance on how to partially satisfy European Union regulations (EC No 482/2008 [EU08])

The scope of the document is defined as all software components across their overall lifecycle within the ANS system. Furthermore the guideline is limited to the ground segment of air navigation services and explicitly excludes aircraft software. A key element of the document is software safety and therefore all references made to software lifecycle data are to be understood in the context of safety assurance.

Figure 7 shows the various levels of guidance which are provided by EUROCAE ED-153 [EUROCAE09]. At the top there are two relevant regulations imposed by the European Union, which cover the contexts shown in Table 2 [EU04, EU08].

Regulation	Context
(EC) No 552/2004	Interoperability of the European Air Traffic Management Network
(EC) No 482/2008	Establishing a Software Safety Assurance System to be Implemented by Air Navigation Service Providers

Table 2: European Union Regulations partially satisfied by Guidance of ED-153 [EU04, EU08]

Chapter 3 of ED-153 provides guidance on how to set up and operate a software safety assurance system (see chapter 3.3). Chapters 4 to 7 of ED-153 deal with the primary, supporting and organisational life cycle processes and additional objectives of the process (see chapter 3.4). This is depicted in Figure 7, where the software development products (the inputs) are transformed in the lifecycle process to software safety assurance products (the outputs). The requirements and recommendations demanded by the guidelines in these chapters are provided in full detail in Annex A of this thesis.

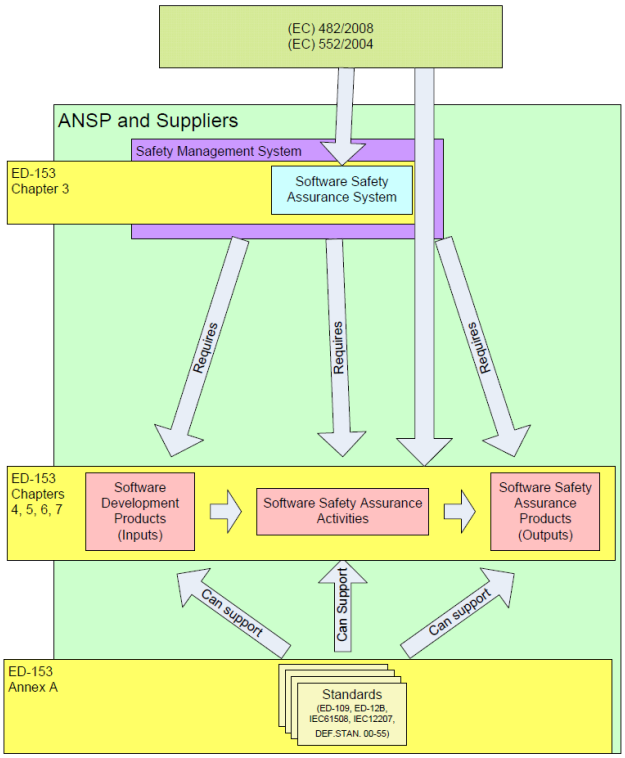

Figure 7: Levels of Guidance provided by ED-153 [EUROCAE09]

3.2 Software Assurance Level

ED-153 introduces the software assurance level (SWAL) as a strategic management method which is used for allocating the appropriate effort that should be spent on safety assurance per software component. Hence the software assurance level is an assessment procedure for software components to define the recommended rigour of the assurance process throughout the whole lifecycle. The rigour in generating the assurance evidence should be in line with the risk presented by the software.

The allocation is done on the basis of the likelihood of software malfunction and the severity of the consequences caused by these malfunctions (see also chapter 2.1.3 for the definition of hazards). The SWAL does not replace the safety requirements for the software; it is itself one of the requirements. To be compliant with the SWAL level, the

software supplier has to take systematic actions to ensure that sufficient evidence of the product and process is available. It has to provide the evidence that its software meets an appropriate level of confidence and assurance in order to contain the risk presented by the system. [EUROCAE09]

ED-153 offers four levels of software assurance, where the first level (SWAL 1) is the most rigorous one, followed by three levels where the rigour decreases from level to level. Table 3 shows the dependency between the likelihood of a consequence (in ED-153 the consequence is called the “effect” [EUROCAE09]) and its severity. For detailed information on severity classes, likelihoods and examples please refer to the official ED-153 standard [EUROCAE09].

Effect Severity Class / **Likelihood of generating such an effect**	**1**	**2**	**3**	**4**
Very Possible	SWAL 1	SWAL 2	SWAL 3	SWAL 4
Possible	SWAL 2	SWAL 3	SWAL 3	SWAL 4
Very Unlikely	SWAL 3	SWAL 3	SWAL 4	SWAL 4
Extremely Unlikely	SWAL 4	SWAL 4	SWAL 4	SWAL 4

Table 3: Allocation of SWAL Levels in accordance with Effect Likelihood and Severity [EUROCAE09]

In comparison with SIL (see chapter 2.1.5), the SWAL is more focused on software. The determination does not depend on availability and reliability, but rather on quantitative and qualitative effects that cause harm to humans or property. Figure 8 provides a mapping of the assurance levels of IEC 61508 to those of the ED-153 guideline. Compliance details can be found in Annex A of the ED-153 guideline.

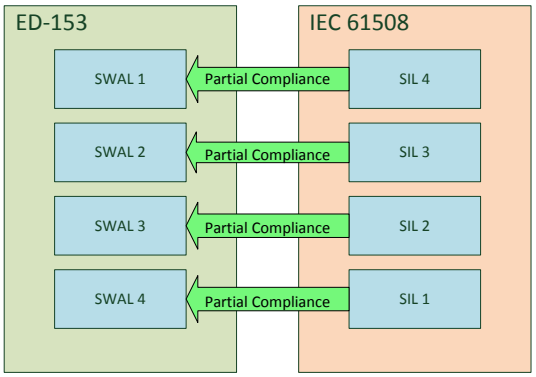

Figure 8: Mapping of SWAL (ED-153) to SIL (IEC 61508) [EUROCAE09]

3.3 Software Safety Assurance System

EUROCAE ED-153 [EUROCAE09] defines a complete software safety assurance system, comprising overall objectives and a software safety process with the following stages:

- Assessment Initiation
- Assessment Planning
- Requirements Specification
- Assessment Validation, Verification & Process Assurance
- Assessment Completion

Due to the fact that ED-153 [EUROCAE09] primarily addresses Air Navigation Service Provider (ANSP), some of the described objectives and tasks within the safety process are not relevant for the software supplier. Annex B of the guideline provides three different role and responsibility scenarios to fulfil the above-mentioned objectives and to achieve conformity with the defined safety process. The least appropriate one is the scenario, where the ANSP internally performs the software development. The other two scenarios differentiate between the delivery of a major system and delivering only software. There are only three extra processes which have to be carried out by the software supplier, when it delivers a system rather than only software:

- Process of showing the isolation of software components
- Process for initiating the software safety assessment
- Process for analysing the system requirements and system architectural design

Based on its focus, the thesis will further concentrate on the most comprehensive scenario, in which the software supplier delivers an equipment part of a system which fulfils the requirements of the ANSP. The context for this scenario is also applicable to the lifecycle processes which are discussed in chapter 3.4.

3.3.1 Software Safety Assurance System

Table 4 shows the objectives of the software safety assurance system which are applicable for the software supplier. Those tasks in which the software supplier actually takes the lead are marked in green. These objectives are derived from the requirements of the commission regulation (EC) No 482/2008 [EU08] and shall ensure the assignment of responsibilities to the ANSP and the software supplier. This regulation sets only high-level criteria which are covered by the objectives of the individual life cycle processes as well (see chapter 3.4). Due to this fact they are only described by their tasks and are not mapped within the objective mapping process in chapter 6. [EUROCAE09]

Legend: A … Accept; C … Contribute; L … Lead

Objective	ANSP	SW Manufacturer	Objective N° ED-153
Implementation	L	C	3.0.1
Requirements Correctness and Completeness	A	L	3.0.2
Requirements Traceability Assurance	A	L	3.0.3
Unintended Functions	A	L	3.0.4
SWAL Allocation	L	C	3.0.5
Requirements Satisfaction Assurance	A	L	3.0.6
Configuration Management Assurance	A	L	3.0.7
Assurance Rigour Objective	L	C	3.0.8
SWAL Assurance	C	L	3.0.10
Software Modifications	A	L	3.0.12
COTS (Commercial Off The Shelf)	A	L	3.0.13
Isolation		L	3.0.14
All On-line Aspects of SW Operational Changes	L	C	3.0.15
Argument Production	L	C	3.0.17

Table 4: Software Safety Assurance System Objectives [EUROCAE09]

The tasks within these objectives that are of importance for a software supplier are [EUROCAE09]:

- Requirements correctness and completeness shall ensure that there is a correct and complete statement of what is required by the software.
- Requirement traceability assurance shall ensure that requirements are traced to the level required by the SWAL.
- Unintended functions shall ensure that software implementation does not contain functions, which may affect safety.
- Requirements satisfaction assurance shall ensure that the software satisfies its requirements by a defined level of confidence.
- Configuration management assurance shall ensure that all assurances are derived from its dependencies.
- SWAL assurance shall provide confidence based on arguments and evidence as defined by SWAL.
- Software modifications shall ensure that software changes lead to a re-assessment of the safety impact and SWAL allocation.
- COTS shall ensure that the same level of confidence is provided for software products that are not developed by the software supplier.
- Isolation shall ensure that software, which cannot be isolated, has the same SWAL as the most critical component allocated.

3.3.2 Software Safety Assurance Process

In addition to the previously shown objectives of the software safety assurance system (see chapter 3.3.1) there are also roles and responsibility recommendations for the

software safety assurance process, which are defined in Table 5 [EUROCAE09]. These objectives are mapped within the objective mapping process to the integrated process model (see chapter 4) in chapter 6 and Annex A.

Legend: A … Accept; C … Contribute; L … Lead

Objective	ANSP	SW Manufacturer	Objective N° ED-153
Software Safety Assessment Initiation			
System Description	C/A	L	3.1.1
Operational Environment	C/A	L	3.1.2
Regulatory Framework	C/A	L	3.1.3
Applicable Processes and Guidance	C/A	L	3.1.4
Risk Assessment and Mitigation Process Output	L	C	3.1.5
Software Safety Assessment Planning			
Software Safety Assessment Approach	C/A	L	3.2.1
Software Safety Assessment Plan	C/A	L	3.2.2
Software Safety Assessment Plan Review	C/A	L	3.2.3
Software Safety Assessment Plan Dissemination	C/A	L	3.2.4
Software Safety Requirements Specification			
Failure Identification	C/A	L	3.3.1
Failure Effects	C/A	L	3.3.2
Assessment of Risk	L	C	3.3.3
Software Requirements Setting	C/A	L	3.3.4
Software Safety Assessment Validation, Verification and Process Assurance			
Software Safety Assessment Validation	C/A	L	3.4.1
Software Safety Assessment Verification	C/A	L	3.4.2
Software Safety Assessment Process Assurance	C/A	L	3.4.3
Software Safety Assurance	C/A	L	3.4.4
Software Safety Assessment Completion			
Document Software Safety Assessment Process Results	C/A	L	3.5.1
Software Safety Assessment Documentation Configuration Management	C/A	L	3.5.2
Software Safety Assessment Documentation Dissemination	C/A	L	3.5.3

Table 5: Software Safety Assessment Process Responsibilities [EUROCAE09]

3.4 Lifecycle Processes

In addition to the objectives for a software safety assurance system (see chapter 3.3), EUROCAE ED-153 defines objectives for the primary, supporting and organisational lifecycle processes [EUROCAE09], which are listed in Table 6. Some of these sub-processes are considered as out of scope for this thesis, which mainly focuses on the primary process development and its supporting processes. These considered processes

are also in line with the integrated process model ISaPro® (see chapter 4), which is the basis for the mapping process of chapter 6.

Process	Applicable
Primary Lifecycle Processes	
Acquisition Process	
Supply Process	
Development Process	✓
Operation Process	
Maintenance Process	
Supporting Lifecycle Processes	
Documentation Process	
Configuration Management Process	✓
Quality Assurance Process	✓
Verification Process	✓
Validation Process	✓
Joint Review Process	✓
Audit Process	
Problem Resolution Process	
Organisational Lifecycle Processes	
Management Process	✓
Infrastructure Process	
Improvement Process	
Training Process	

Table 6: Lifecycle Processes of ED-153

4 Integrated Process Model

In order to be compliant with safety standards, various approaches to process-oriented models and lifecycles are described in literature. Some of these approaches are lifecycle models that are derived from several standards (see chapter 2.3); some of them are procedures that are proposed by different authors (e.g. [Sto96]). In addition, further standards are demanded by industry such as CMMI (Capability Maturity Model® Integration) or SPICE (Software Process Improvement and Capability Determination).

In order to handle all the required standards, the Vienna Institute for Safety & Systems Engineering (VISSE) developed an integrated process model called ISaPro®. It provides a framework for meeting the criteria of the required standards in one process model. Furthermore, ISaPro® provides an approach for meeting the objectives in a systematic way. This ensures that the required activities are done at the right time in the right phase. [TKH12]

The approach consists of three in-parallel, well synchronised lifecycles and their supporting processes. ISaPro® includes all the necessary disciplines for developing a safety-critical system. By synchronising the individual processes, this framework further ensures that all lifecycles have the necessary interdependencies between them. On the timeline the whole procedure model is divided into four sections that are called “spaces” [TKH12]. The problem space is performed in the pre-project phase; the modelling and solution spaces are executed during the development project. After completion of the development project, specifically the system development, the model is continued in the form of system maintenance in the operation space. [TKH12]

Figure 9 depicts how the different lifecycles are synchronised over time within the four defined spaces. If the targeted system is complex, further analyses have to be done. To break down the software part into software components, the light-coloured processes within the solution space are necessary.

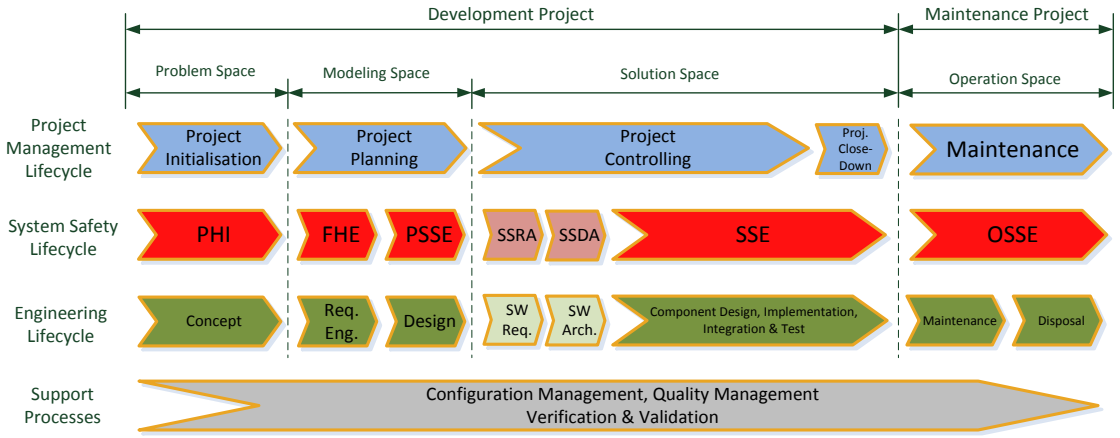

Figure 9: Adapted ISaPro® Framework [based on TKH12, TSS12]

During the problem space, the main aim is to identify the costs and the lead time of the project. One of the main cost drivers in safety-critical development projects is the targeted SIL (see chapter 2.1.5), which is identified in the preliminary hazard identification (PHI). Within the modelling space the requirements and design of the system have to be thoroughly defined. Based on those definitions, the safety objectives and derived requirements are identified in the functional hazard evaluation (FHE) and preliminary system safety evaluation (PSSE) processes. These outputs are considered in the project planning process, where the magic triangle (time, cost and scope) of the project is defined [CG06]. The solution space ensures that all safety objectives and requirements will be fulfilled by coordinating the planned activities. After the development of the system is completed, usually the phase of operation starts. Due to the focus of this thesis on the development of systems, this phase will be considered as out of scope. [TKH12]

The following sections will briefly describe the main aims and activities of each lifecycle and how they interact with each other as a full framework in order to ensure the safety goals are met. The description starts with the project management lifecycle, followed by the engineering lifecycle, which is necessary to deliver inputs for the safety activities, described in the processes of the safety lifecycle. The chapter concludes with a brief introduction to the support processes that facilitate the three lifecycles.

4.1 Project Management Lifecycle

In accordance with the focus on development, the project management lifecycle consists of four processes: the initialisation, the planning, the controlling and the close-down processes. The use of project management is necessary for transforming a complex project into manageable activities in order to ensure that the magic triangle is balanced all the time. [Gar06]

The project initialisation process includes the determination of the scope of the project, a preliminary budget and a time schedule. These outputs are heavily dependent on the pre-estimated SIL, which is identified in the PHI, and therefore must include all prospective safety activities [TKH12].

The planning process comprises a detailed plan that includes all the work packages within the project lifecycle. These packages are again dependent on the required safety activities, which were defined during the safety planning process [TKH12]. In addition, the parallel defined system requirements and design will further increase the accuracy of the project plan.

The controlling process includes all the activities necessary to manage deviations of the prospective plan. It consists of periodically planned controlling meetings, in which the planned activities are compared with the current status of the project. In the case of deviations, steering measures have to be arranged in order to keep the project on track.

The project close-down process ensures that the outstanding work is done, the project documentation is finished and the scope for the post-project phase is defined. In addition, the project team has to transfer all the lessons learned into the line organisation. Finally the team itself has to be dissolved.

4.2 Engineering Lifecycle

The engineering lifecycle of the ISaPro® integrated process model consists of the creation of a concept, the requirements engineering, the design, the realisation and the operation phase (see Figure 9). This thesis focuses on the planning and development phases, excluding the processes of maintenance and disposal. The processes are based on traditional system development approaches such as the V-model (see chapter 5.5.1).

The technical concept provides a rough system design, based on the available information supplied by the customer or the project owner. It depicts the technical realisation, which is a mixture of the customers' needs and their technical solutions. [TSS12]

The requirements engineering phase is the basis for the project. It defines what stakeholders expect from the system in order to meet their needs [HJD10]. This includes first of all the functional requirements, which define what set of functions the system should offer. In addition to that, the non-functional requirements also have to be defined which describe how the system should achieve these predefined functions. These non-functional requirements could be, for example, requirements regarding availability, performance or even constraints. One of the most important non-functional requirements class is the class of the safety requirements, which will be gathered during execution of the FHE (see chapter 4.3.2) [TKH12]. Especially during the development of safety-critical systems or applications they should be treated in a very particular way: apart from being identified and defined they should furthermore be traced throughout the whole lifecycle [HJD10].

These defined requirements are then inputs for the system design. The aim of this process is to define a technical solution which meets all requirements including the important safety requirements.

Before starting the realisation process, further tasks must be carried out in advance, especially when a complex system which needs further detailed analysis is identified. In such a situation there is a need for requirements engineering and design on a component level [SW01]. These tasks can also be seen as parts of the realisation process, but in the case of a software system there are designated processes called "software requirements engineering" and "software design" [TKH12]. They basically include the same tasks as the system level, but within the scope of software.

The final process within the solution space is the realisation, which consists of several sub-processes such as the optional component design and the real implementation,

followed by integration and final tests [TKH12]. Component design is, as previously mentioned, the process of further dividing the system into parts in order to manage very large and complex systems more easily. Implementation is the conversion of the planned ideas into a working system – in the case of software, this is the coding itself. During integration, the divided parts of a system (e.g. hardware and software) are put together in order to assemble the functionally working system. The test sub-processes are the verification of the predefined requirements in order to ensure that the system is working correctly according to the specifications.

Within the whole engineering lifecycle, traceability is of utmost significance. Therefore all requirements have to be traced to documented design decisions and the corresponding verification and validation activities. When performing detailed analysis on the software level, the tracing from system level to software level is of the utmost importance.

4.3 Safety Lifecycle

The safety lifecycle ensures compliance with the required safety standards such as IEC 61508, ISO 26262 or EUROACE ED-153 by including their required activities into the processes of the lifecycle.

As Figure 9 shows, the safety lifecycle consists of five main and two optional processes. The lifecycle starts with the preliminary hazard identification (PHI) followed by the functional hazard evaluation (FHE) and the preliminary system safety evaluation (PSSE). When some system complexity is met that needs more detailed analysis on the component level, the two optional processes have to be performed. In the case of software, these additional software safety lifecycle processes are the software safety requirements analysis (SSRA) and the software safety design analysis (SSDA). These processes are similar to the FHE and the PSSE, but within the scope of software only. The system safety evaluation (SSE) is the subsequent process within the lifecycle, and is performed until the development of the system has finished. After the system has gone into operation, the operational system safety evaluation (OSSE) is initiated. Due to the focus of this thesis, this last process is considered as out of scope. [TKH12, TSS12]

Each of the main safety processes answers a different question regarding safety, as shown in Figure 10:

Figure 10: Questions of the Different Safety Processes [based on TS10, TSS12]

The following sections will describe the six processes depicted in Figure 10, which are applicable during the development of the system.

4.3.1 Preliminary Hazard Identification (PHI)

Based on a rough technical concept of the system, designed and developed in the engineering lifecycle, the PHI is started on the top level at a very early stage. This helps in taking appropriate actions such as redesigning the system to reduce or remove hazards or even challenging the whole system concept. The goal of the process is to identify all theoretically potential hazards by knowing the rough concept and main business use cases. This identification could be achieved using widely known methods for idea generation such as brainstorming, reviewing former projects or using checklists. The list of identified hazards should consist of their causes, their possible consequences (see Figure 3), a first risk assessment and presumable risk mitigation strategies. By having the full list of preliminary identified hazards it should be feasible to allocate a SIL to the system. This is done in order to estimate how further detailed safety activities should be executed. [Sto96, TKH12, TSS12]

4.3.2 Functional Hazard Evaluation (FHE)

The aim of this process is to identify all of the system functions which could lead to one or more hazards. Based on the output of the PHI in the form of a preliminary hazard list, and by including the requirements of the system, it is possible to make a statement about how safe the system must be. Possible methods supporting the achievement of this goal are a PHA (see chapter 2.4.1) or a HAZOP (see chapter 2.4.3). [TSS12]

The main outputs of the process are the definitive SIL, an extended list of hazards (which was started in the previous PHI process) and, based on that list, the derived safety requirements for the system. These safety requirements are necessary in order to achieve the safety goals and prevent the identified hazards from occurring or at least to mitigate their risks to an acceptable level. Due to the fact that system requirements depend on safety requirements and vice versa, the system requirements have to be updated when safety requirements are defined. [TKH12, TSS12]

4.3.3 Preliminary System Safety Evaluation (PSSE)

In the PSSE process, the system design is analysed in terms of whether the safety requirements and an acceptable level of risk are met. The process consists of several analyses on the subsystem, component and software levels for the purpose of analysing the interdependencies between them. This is done with the help of methods such as FMEA (see chapter 2.4.2) and/or FTA (see chapter 2.4.4). [TS10, TSS12]

A further part of the investigation is the evaluation of the system design itself and the question of whether additional hazards are raised by that design. This will probably result in new, derived safety requirements, which might necessitate an update of the safety requirements and the system design itself. If so, the PSSE has to be repeated until the

system design meets all safety requirements and does not raise any new hazards. In addition to the definitive set of safety requirements, a preliminary safety case is created. [TKH12, TSS12]

4.3.4 Software Safety Requirements Analysis (SSRA)

The SSRA inspects all software and interface requirements to detect faults and defects, which could lead to software hazards. As well as the software parts, all hardware parts which could trigger software malfunction are also taken into account. Additionally, all software-related safety requirements are verified in terms of their correctness and completeness. To identify all relevant safety-critical functions at an early stage, a Safety Critical Function List (SCFL) is created on the basis of predefined objective criteria. [TSS12]

The main aim is to identify software safety requirements in order to meet the required SIL. In addition, all the safety-critical functions of the software have to be documented.

4.3.5 Software Safety Design Analysis (SSDA)

The SSDA is, like the PSSE but on the software level, the verification of whether the software safety requirements are adequately covered in the software design. Additionally, the analysis should detect whether the software design introduces new hazards. Common methods for supporting this process are FMEA and FTA, which are especially tailored to software and therefore performed on a qualitative level. [TSS12]

4.3.6 System Safety Evaluation (SSE)

The SSE is the last safety process within the development of the system and is done in parallel with the detailed design and implementation. This process should ensure that previously defined safety requirements and the design aspects have been correctly implemented in such a way that the remaining system risks are below an acceptable level. In order to prove this correctness of implementation, verification methods and tests are used. The safety case which was started in the previously process has to be continued using all available information. [TSS12]

The SSE is performed periodically during the whole system lifecycle, whereas the process is often called the “operational system safety evaluation” (OSSE) after the initial development has finished [TSS12]. This process is closely connected to the safety case document, which should be kept up to date until the disposal of the system, especially when changes are made to the system [TSS12].

4.4 Support Processes

ISaPro® recommends having at least three support processes including configuration management, quality assurance and verification and validation. In specific cases, change

management is considered as a separate process, but it can also be part of the configuration management [CMMI10]. These processes support all three lifecycles by ensuring the integrity and quality of work.

The exact purpose of configuration management is to achieve and ensure the integrity of work through several tasks. The first task is to identify the work products which should be put under configuration management. During the lifecycle of the project, changes to those items have to be controlled, documented and reviewed. Work products can be any output within the whole project lifecycle, e.g. plans, specifications, requirements, documented design, code, etc. In the majority of the cases, at given points in time, e.g. after a review, a work product is marked as a valid baseline (release). This indicates that this version of the work product is a stable basis for continuing evolution of the configuration item. [CMMI10]

Quality assurance consists of two main tasks: the monitoring of the processes and the evaluation of their effectiveness. This can be achieved by reviewing all defined work products and processes on a regular basis. It has to be defined which of the processes and work products should be reviewed and also how and when the review should take place. Of particular importance in the definition of work products and processes are the safety-critical ones. All tasks belonging to quality assurance have to be performed either by an independent department or at least an independent person. [TSS12]

The last support process is the verification and validation. Verification ensures that the system meets its requirements. This is done with the help of reviews, inspections, static code analysis and tests. Validation ensures that the system fulfils its intended use as expected by the stakeholders and regulations. This also includes the proof that all specifications required by the safety standards are fulfilled. [TSS12]

5 Agile Software Development Methods

In the late 1990s, new approaches to software development started to emerge as alternatives to the traditional development methods (described in chapter 5.5.1) which were widely used at that time. The first representatives of these so-called agile methodologies introduced completely new ways of approaching software development.

> *"Agility means that you are faster than your competition. Agile time frames are measured in weeks and months, not years."*
>
> *Michael H. Hugos [Hug09]*

Although different approaches were introduced by different people, they were all based on similar beliefs and ideas. The central topic was the creation of a new, more efficient way of developing software in terms of a lightweight and more flexible process. At the same time, this process should ensure high quality software products. The term >agile< was adopted as a kind of umbrella term for all those new approaches which fitted in with the common values and principles, which are introduced in chapter 5.1. The best-known approaches which put these values and principles into practice are described in chapter 5.3.

Nowadays, agile methods are widely accepted as a development approach for software [VerOne13]. A German study has pointed out that the majority of companies use either traditional or agile approaches, or even hybrid forms depending on the kind of project that has to be accomplished [Kom12]. This course of action makes the most of the advantages of all the various traditional and agile approaches. There are also empirical data and case studies available on their benefits compared to traditional methods. The most applicable ones are presented in chapter 5.4.

5.1 Values and Principles

In February 2001, seventeen independent representatives of various software development and programming methodologies committed themselves to four core values, called the "Agile Manifesto" [AgiAll12]. These values are supplemented by twelve principles which further explicate what it means to be agile. Both the values and principles formed the basic framework for the birth of the Agile Alliance and are still valid nowadays. This Agile Alliance was founded at that meeting. It is a non-profit organisation which has committed itself to advancing agile development principles and practices. [AgiAll12]

5.1.1 Values

Manifesto for Agile Software Development:

"We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions *over processes and tools*
Working software *over comprehensive documentation*
Customer collaboration *over contract negotiation*
Responding to change *over following a plan*

That is, while there is value in the items on
the right, we value the items on the left more."
Ward Cunningham [Cun01]

The introductory paragraph of the quote makes it clear that this group of people were experienced software development practitioners who thought that they had discovered some new and probably better ways of developing software. This first paragraph is followed by the four main principles which prioritise values. The concluding paragraph is intended to indicate that the values on the right cannot be ignored completely. Rather, it points out, there has been a shift of priorities in favour of the left-hand values.

Individuals and interactions *over processes and tools*

The first main value is that individuals and interactions are valued more than processes and tools. This should indicate that these new kinds of methodologies focus on people and not explicitly on their roles as stated in the organisation chart. Team members should use processes and tools as aids to leverage the effects of agile development. Another important fact is that people are not as easily exchangeable as other resources such as infrastructure and similar elements. The direct and verbal interactions between well-educated and trained team members are necessary for new and sophisticated solutions where all dependencies are considered. [Coc06, WB10]

Working software *over comprehensive documentation*

The second value points out that a working system is ruthlessly honest and therefore shows exactly, what has been achieved so far. Working software can even be deployed and operated by the customer, while documentation is primarily an aid for the team members to specify the unreliable future as well as possible [Coc06]. It does not tell the user anything about the progress of development and therefore only as much documentation as is actually necessary should be created.

Customer collaboration *over contract negotiation*

This third core value should usher in a new era in which the customer has an amicable relationship with the contractor beyond organisational boundaries. This is indicated by the customer having a voice in joint decision-making and an involvement in development planning and the approval of the recently delivered work. Contracts are of course useful, but experience shows that in most cases the system specified in the contract does not exactly correspond to the system the customer and particularly the end-user needs [WB10].

Responding to change *over following a plan*

The final value addresses the need to respond to changes as fast as possible. Creating and referring to a plan is reasonable; e.g. each agile method has a development planning phase, but the plan has to contain mechanisms for dealing with changing priorities [Coc06]. It simply does not make sense to follow a plan when it is clear that it cannot be met anymore.

5.1.2 Principles

In addition to the core values of the agile manifesto (see chapter 5.1.1), the practitioners of agile software development committed themselves to twelve principles, which indicate what it is to be agile.

This master's thesis will not list all of them in complete detail; rather it will concentrate on the basic statements based on the principles of the Agile Alliance [AgiAll12]. The all-embracing statement of principles can be found on the website of the agile manifesto [Cun01].

One principle is to satisfy the customer by means of an early and frequent delivery of valuable software. In order to achieve this statement, two more principles have to be considered. One is about working software itself as the main measure of progress and the second one is about the frequent delivery of that working software. Basically, the combination of all three principles should enable early feedback from the customer or end-users in order to adjust development based on those inputs to achieve the main aim of customer satisfaction.

This leads directly to another principle, whereby the agile process welcomes changing requirements, even late in development. Without this principle, customer feedback cannot be integrated into the software. There have to be mechanisms available for dealing with changing priorities.

These principles are followed by others which deal with individuals. The first is that business people and developers should work together on a daily basis within the scope of the project. This is emphasised by the statement on the most efficient and effective method of conveying information to and within the project team: the face-to-face conversation. Another principle of dealing with people is that the project should be built on motivated individuals. This can be achieved by providing them with the necessary environment and support, and the trust that they will get the job done.

In addition, there are some statements about the development techniques themselves. One is that the best architectures, requirements and designs emerge from self-organised teams. This indicates on the one hand the responsibility of the team and on the other hand the fact that architecture, requirements and design are not made prior to implementation, but rather they emerge during this process. Another principle is that continuous attention should be paid to technical excellence, because only good design

enhances the agile approach. In order to accomplish this excellence, some common technical practices have become accepted in various agile approaches (see chapter 5.2).

Then there is a principle regarding the promotion of a sustainable development so that sponsors, developers and users are able to maintain a constant pace. This statement deals with the term project efficiency. It indicates that long hours of work by the team should be avoided because it will make them tired, which in fact will lead to more errors in their work and thus a reduced pace in the following period [Coc06].

Another principle deals with the term simplicity. This is defined as the art of maximising the amount of work not done [Cun01]. This has to be achieved within the boundary condition of delivering valuable and qualitative software. The challenging fact is that making things simple is quite difficult, as Pascal noticed in the 17th century:

> *"I have made this letter longer than usual, only because I have not had time to make it shorter."*
>
> *Blaise Pascal [translated from French]*

The concluding principle is about the importance of reflecting on how to become more effective at regular intervals. This should ensure that behaviour is adjusted accordingly via of the reflection work of the team. It is the basic requirement for a continuous improvement process which evolved from the Japanese term "Kaizen" and is nowadays part of almost every quality management system [Ima86].

5.2 Technical Practices

Some technical practices have become accepted across various approaches (see chapter 5.3) because of their support for a lean and agile process. This chapter provides an overview of how to be agile on the operative basis of the process.

5.2.1 User Stories

In the agile process, requirements are necessary to estimate the effort required in order to give the stakeholders the opportunity to prioritise depending on those estimated values. Furthermore, requirements should ensure clarity between the customer and the developers. In order to do so, the requirements are written as user stories. [WB10]

In most cases, user stories follow structured templates, like these [Coh09]:

> *"As a <type of user>, I want <some goal> so that <some reason>."*
>
> *"In order to <achieve value>, as <type of user>, I want <some goal>."*

These user stories should be written initially by the user itself or at least by "*user proxies*" who represent the customer or the end-user in the project [Coh04]. Such persons can be

end-users, managers, salespersons, domain experts or business analysts. Afterwards, the user stories are improved in a discussion process including the customer and the contractor by adding necessary details and remarks [WB10].

5.2.2 Test-driven Development

Another technical practice is test-driven development (TDD). In traditional software development approaches, each part of the software is usually designed first, then coded and finally tested during the verification phase. When defects are detected in this phase, the code is corrected in order to repair the defects that occurred. Then these changes are verified again.

The process of test-driven development turns this way of thinking around and starts with the implementation of a test in order only to write the code, which is necessary to achieve the clear goal of passing the test. Design is carried out in the final stage, during which the code is restructured in such a way that the simplest design is the result. This last step is also called refactoring and is explained in chapter 5.2.3. [Kos07]

This previously described cycle, which is depicted in Figure 11, is repeated for every test that has to be implemented.

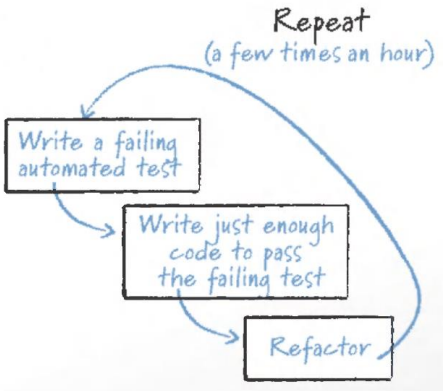

Figure 11: Test-driven Development Cycle [Coh09]

This process automatically results in comprehensive code coverage through automated tests, which further helps to avoid defects in the code when adding new functionality [Kos07].

5.2.3 Refactoring

As stated in chapter 5.2.2, refactoring is the last stage within the test-driven development cycle. Refactoring is the process of changing a software system without altering the external behaviour of this part of the code in order to improve its internal structure. It is a way to minimise the risk of new defects occurring in the future by cleaning up the code. This process of cleaning up the code ultimately results in the final design of the software component [FBB+99]. A basic prerequisite for refactoring is high code coverage provided

by unit tests in order to be sure that the software system is running as expected after the redesign [Kos07].

According to Martin, refactoring also helps to prevent *"code rot"*, which is a term for a typical syndrome within software where the code is allowed to decay until the decision is made that it has to be completely rewritten [Mar08].

5.2.4 Evolutionary Design

This technical practice is more of an approach and is basically the result of using test-driven development (see chapter 5.2.2) and refactoring (see chapter 5.2.3). The idea is to evolve the design as new requirements arise. Therefore software coding is initiated with a simple design, e.g. by using TDD. Afterwards this design is changed only when requirements force this step. Necessary changes are then achieved by refactoring and automated tests. [Els07]

5.2.5 Continuous Integration

Continuous integration is an extension of the use of nightly builds [Coh09]. The basic idea is that the written code is checked into the source code repository a few times a day in order to ensure that software integration is continuously tested by automated integration builds. It is important that all procedures after the check in, e.g. building software, running automated tests and sending notifications, are executed automatically. [DMG07]

5.2.6 Pair Programming

Pair programming (PP) is the term for a practice whereby two developers sit next to each other at one machine. It is a dialogue between two people simultaneously implementing the requirements including analysis, design, coding and testing [BA04]. This method should not be used all day long; it should rather be used for complex and risky parts of the software project [Coh09]. A further recommendation is to switch partners frequently in order to transfer knowledge throughout the team. Studies have shown that pair programming slightly increases costs, but demonstrably contributes towards enhancing quality [DAS+07].

5.2.7 Collective Code Ownership

The basic principle of collective code ownership is shared code. It basically states that the entire team owns the whole source code and therefore is responsible for it. Everyone can change code in any part of the system at any time. Automated tests ensure that the correct operation of the code is not affected by a developer changing the code who is unfamiliar with the software module. This mutual trust ensures that a developer is able to do all the tasks which are necessary to accomplish the user story. [BA04, Els07]

5.3 Approaches

Various approaches have emerged in connection with those values, principles and technical practices during the last 15 years. Extreme Programming (XP) was probably one of the first approaches in agile software development, whereas today the most widespread approach is definitely Scrum [BN07, VerOne13]. This is particularly based on the fact that Scrum is very management-oriented. Therefore a lot of companies use variants of or approaches based on Scrum [BN07] or even include other practices like XP in the Scrum process. These two approaches are briefly described in the following two sections.

Other relevant approaches [VerOne13] that will not be introduced in this chapter are feature driven development (FDD) [CLD99, Ric12] and the software variant of Kanban [And10], a process which originated in automotive production [GM03].

5.3.1 Extreme Programming (XP)

Extreme Programming consists of a collection of the most successful practices and was first introduced by Kent Beck in 1999 [BA04]. In addition to the practices, this approach also describes their interdependencies based on the agile values [WB10].

The approach is based on values and principles which are basically in line with those of the agile manifesto (see chapter 5.1). These values should be implemented by primary and corollary practices. Due to the fact that the approach needs to be described briefly here, only the primary practices are listed. For further details refer to [BA04]:

- *Sit Together*
- *Whole Team*
- *Informative Workspace*
- *Energized Work*
- *Pair Programming*
- *Stories*
- *Weekly Cycle*
- *Quarterly Cycle*
- *Slack*
- *Ten-Minute Build*
- *Continuous Integration*
- *Test-First Programming*
- *Incremental Design*

Kent Beck [BA04]

5.3.2 Scrum

In contrast to Extreme Programming (see chapter 5.3.1), Scrum is a generic organisational management approach. The process description gives no specifications or guidelines on how to design, code and implement software, and basically Scrum can be used for non-software projects as well [Coh09]. Because of its freedom in terms of technical practices, Scrum is often combined with other agile methods such as Extreme Programming or Kanban. This latter combination is also called “Scrumban” [Lad09].

Scrum arranges all its practices around an iterative, incremental process skeleton as can be seen in Figure 12. In the first step, the role of the product owner creates a prioritised list of user stories which is called the “product backlog” [Sch04]. During sprint planning, the planning for the next iteration, the team pulls a subset of these stories – which they think that they will be able to accomplish within the next iteration – from the top of the product backlog. This subset is called the “sprint backlog” [Sch04]. During this sprint, which is usually a fixed time box of one to four weeks, a daily scrum meeting is held in order to check the status of the team. Along the way the role of the so-called Scrum master, who is a kind of method coach, keeps the team on track. At the end of the iteration the product should be potentially shippable. The process is formally closed by a sprint review where the accomplished user stories are demonstrated and a team retrospective takes place. After the iteration is finished, the next one starts with sprint planning and so on. [Sch04, ScrAll12]

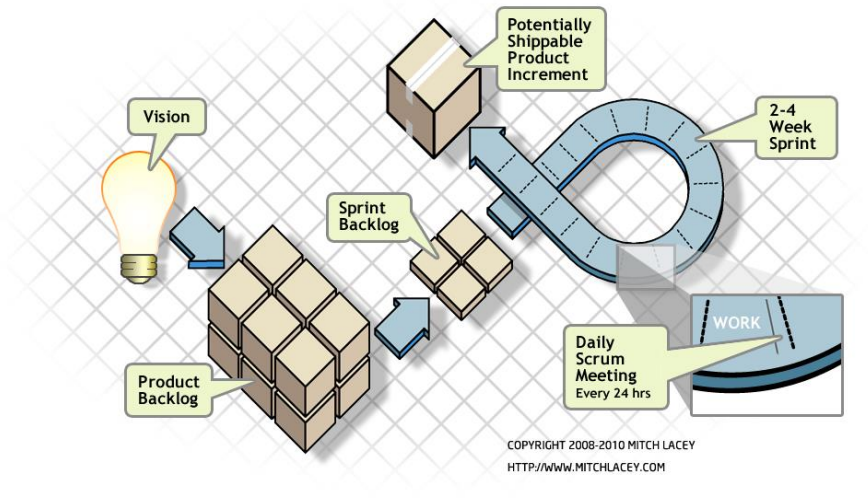

Figure 12: Scrum Framework [Lac12b]

5.4 Scientific Research

A large amount of scientific research about agile methodologies has been published over the past few years. This chapter will briefly introduce the most significant studies.

One study was carried out by Michael Mah in 2008 [Mah08], in which he made an exhaustive comparison of more than 20 agile projects on more than 7,500 traditionally completed projects, using a contemporary worldwide database. It points out that these agile projects are 16 % more productive and have a 37 % faster time to market at a statistically significant level of confidence. [Mah08]

The second study was published by David Rico, also in 2008 [Ric08]. He carried out a survey based on 69 published academic and research papers to evaluate whether agile methods impact the return on investment (ROI) of a project. The results showed that agile

methods are almost as good as the best traditional models, which according to this study are the Personal Software Process (PSP) and the Team Software Process (TSP). In comparison with heavier traditional methods like the Capability Maturity Model® Integration (CMMI) or ISO 9001, agile methods had a higher average ROI. [Ric08]

While these first two studies focused on whether and how projects were mastered better with the adoption of agile methods, a study by Kruchten [Kru04, Kru10] tried to identify the optimal conditions for succeeding with agile methods. Kruchten describes this ideal context as the *"agile sweet spot"* [Kru04, Kru10]. Based on his experience as a consultant to companies adopting agile methods, he defined this sweet spot by the following criteria [Kru04, Kru10]:

- Co-located small teams of ten to fifteen people to facilitate face-to-face communication
- Customer availability to get fast feedback and decisions in order to increase their satisfaction
- New development or so-called "green field" projects in order to avoid maintaining legacy source code [Kru10]
- Interactive types of applications like business applications (in contrast to embedded real-time systems)
- Low to medium criticality in terms of worst case is losing money (and not to harm humans or property)
- Short lifecycles of weeks to months and not years

This report does not indicate that projects outside the sweet spot would not work, but it might be that those projects face challenges that have to be overcome. Possible solutions could be the adaption or tailoring of agile processes, but in some cases the result will be that agile methods are just not suitable for that particular project. [Kru10]

5.5 Interdependencies with Traditional Approaches

This section is intended to give a brief overview of traditional software development lifecycles and their interdependencies with agile approaches. While chapter 5.5.1 introduces the two most familiar approaches within traditional software development, chapter 5.5.2 identifies approaches targeting the combination of agile and traditional procedure models.

5.5.1 Introduction to Traditional Approaches

Due to the focus of this thesis on safety-related software development, the selected approaches are the waterfall model and the V-model. While agile methods use an iterative approach, these two models use a sequential and plan-driven approach, respectively, to developing software. This difference makes it difficult to compare them. As already highlighted in chapter 5.4 there is no simple answer in finding the best procedure model or approach. All of them have their advantages and therefore each

potential software system has to be evaluated before starting development in order to determine the most suitable approach.

The waterfall model, invented by Royce [Roy70] in 1970, was the very first approach within software development. Royce's model is based on a sequential approach where the completion of one development activity allows its successor activity to begin [Roy70]. Figure 13 depicts the core lifecycle of the model with interactions between consecutive development activities or phases.

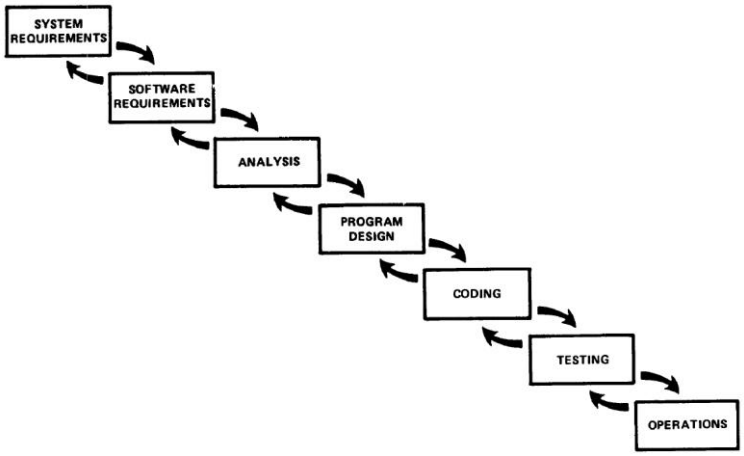

Figure 13: Waterfall Model [Roy70]

The major advantage of this model is the straightforward and very structured approach, which is very efficient when requirements remain relatively stable over the project time [Boe02]. If there are changing requirements this model becomes inefficient in terms of total costs. While the costs of change increase in quite a linear way when using agile methods in combination with test-driven development and refactoring (see chapters 5.2.2 and 5.2.3), the costs increase exponentially over the course of the development lifecycle when using the waterfall model [Els07, WB10]. Figure 14 depicts this diverging development of cost per change.

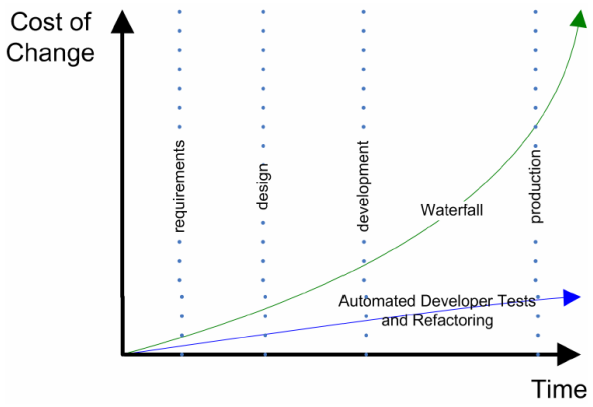

Figure 14: Cost of Change over Time using the Waterfall or Agile Procedure Models [Els07]

The V-model was introduced by Boehm [Boe79] in 1979, just a few years after the waterfall model. The author focuses on the aspects of verification and validation and their context in the software lifecycle. Boehm points out that the initial definition of requirements and design in various levels of detail is linked to verification and validation activities after implementation. Therefore verification and validation activities can be put on the right side of the model in line with the definition on the left side as depicted in Figure 15. [Boe79]

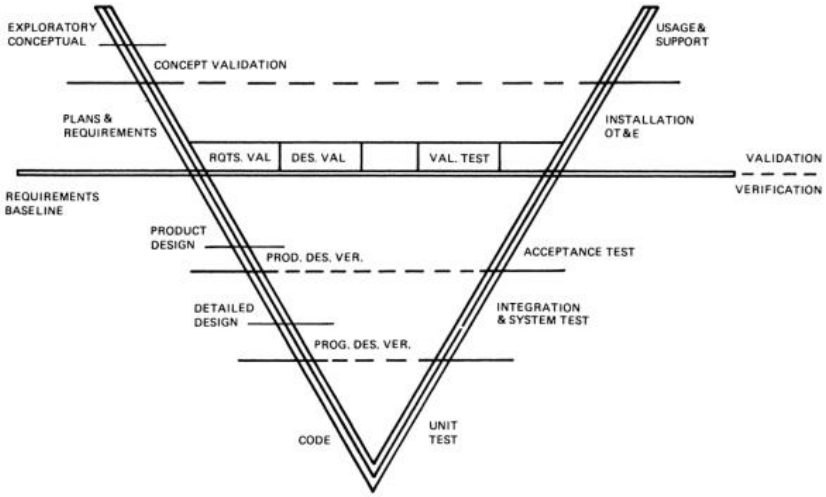

Figure 15: V-Model [Boe79]

Such sequential approaches are also required when dealing with project governance [Coh09], which is used for high-level control of projects in the project portfolio of an organisation. This project overview is often achieved by stage-gate processes, where the software development has to pass various gates along the project lifecycle [Coo08] as depicted in Figure 16. It helps top management to monitor projects effectively, e.g. for forecasting whether a project will exceed its budget or similar issues.

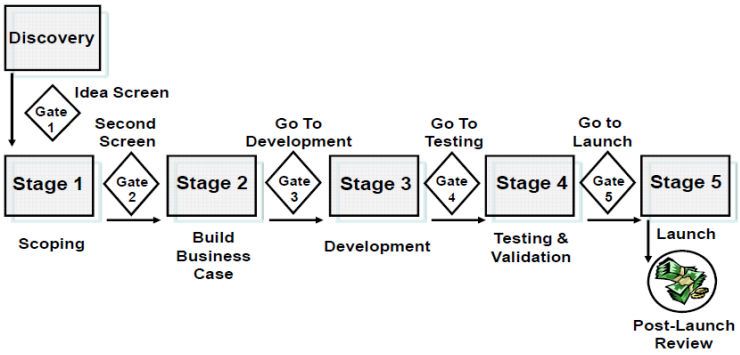

Figure 16: Stage-Gate® Approach [Coo08]

5.5.2 Combination of Agile and Traditional Methods

This section gives a brief overview of how agile approaches are combined with traditional ones in order to meet particular requirements. These adapted procedure models are mainly required by development projects that conflict with agile methodologies. Therefore the development approach has to be adjusted in order to fit the context of the project. When it comes to safety-critical software systems, agile methodologies have to be adapted in order to suit the objectives defined by standard specifications.

The most familiar adaptation of agile procedure models is mixing them with sequential development methods. According to Sliger [Sli06] there are three different scenarios:

- Waterfall-up-front
- Waterfall-at-end
- Waterfall-in-tandem

Regarding the first scenario (waterfall-up-front), there are dozens of reasons why some documents such as a project plan or specifications are needed before starting a project or software development. Some documents might be required by internal parties, e.g. by the management for releasing the budget, or by external parties, e.g. by an authority for confirming compliance to a standard. Regardless of the kind of reasons for which this is necessary, Cockburn [Coc00] recommends generating documentation that is *"barely sufficient"* in order to meet agile values and principles (see chapter 5.1). These upfront tasks can be achieved either ahead of the first iteration or within it as the first backlog item [Sli06]. Besides the fact that this preliminary work provides necessary information for the main stakeholders, it helps the agile team to develop a product vision by compiling the specification [Sli06].

The second scenario (waterfall-at-end) is designed for software projects that need a designated preparation phase in order to achieve tasks that cannot be managed within an iteration. Such a task could be a separate verification and validation by dedicated and independent teams of quality assurance people or even external groups [Coh09]. Another possibility could be an approval process required by an authority or a standard specification [Sli06].

The third scenario (waterfall-in-tandem) is the most complex one, dealing with software development projects that are comprised of more than a single team, using different development approaches [Coh09]. This approach needs a lot of communication and coordination among all teams so that they can pull together. Most likely the largest barriers are the different value sets of the teams, which could regularly result in conflicts [Coh09].

Regardless of which of those three previously mentioned scenarios is used, to be truly agile West [Wes12] recommends pushing the agile approach (light blue activities) as far to the edge as possible, as depicted in Figure 17.

Figure 17: Combination of Waterfall-up-front and Waterfall-at-end [Wes12]

6 ED-153 Objective Mapping

This chapter describes all necessary activities within the development of safety assured software based on the objectives raised by EUROCAE ED-153 [EUROCAE09]. In order to make sure that the objectives of the ED-153 are fulfilled, a mapping of these objectives to the relevant processes of the ISaPro® (see chapter 4) framework is necessary. For this mapping process only the software safety assurance system (see chapter 3.3) and the applicable lifecycle processes (see chapter 3.4) of ED-153 are considered as in scope. The aim of this mapping process is a defined way of working including all necessary activities that have to be conducted in order to be compliant with ED-153.

6.1 Objective Mapping Method

Figure 18 depicts a model of this mapping process. This method should ensure that the overview of the way of working does not get lost due to the multiplicity of various objectives. In addition the result provides a process framework including activities ensuring compliance with EUROCAE ED-153.

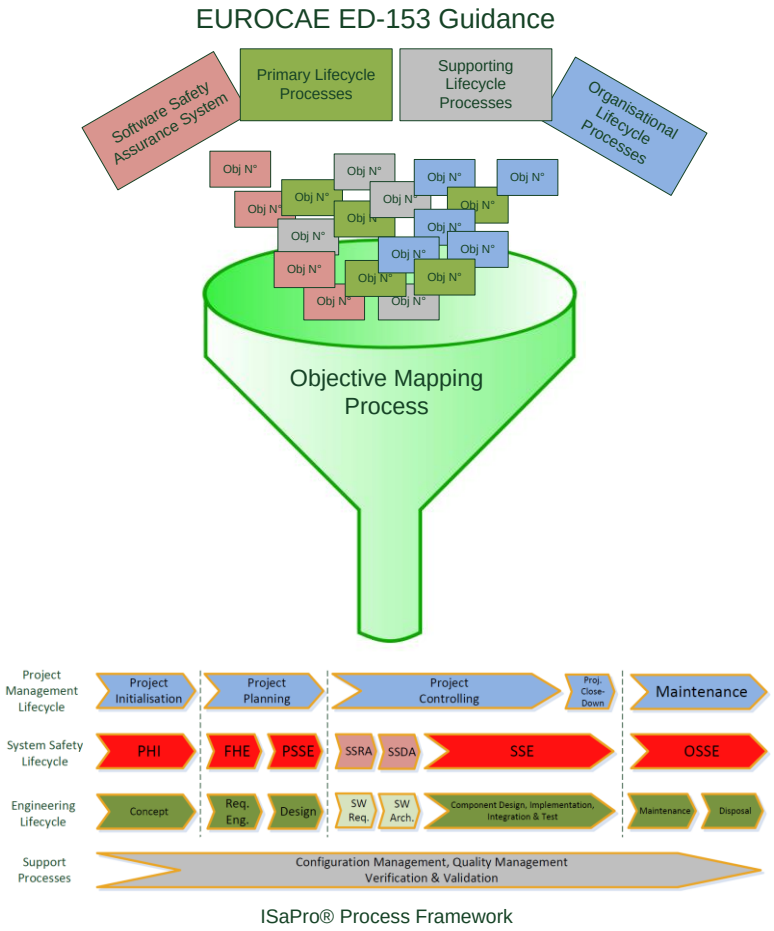

Figure 18: Objective Mapping Process

The realisation of this mapping process is carried out by the following activities:

- First, each activity (or also sub-objective) of the ED-153 objectives is mapped to a process of the ISaPro® process model based on activity comparison. A detailed analysis on which sub-objective is matched to which process is available in Annex A.
- In the case that no activity of the ISaPro® model specifically indicates the activity of the objective, an existing activity will be expanded or a new activity will be created (see comment columns in Table 10, Table 11, Table 12 and Table 13 in Annex A).
- Creation of compacted tables for each category of lifecycle processes (safety assurance, primary, supporting and organisational processes) (see summary chapters in Annex A).
- Creation of a comprehensive list of activities per process of the ISaPro® framework which are necessary to achieve the whole number of objectives (see Annex B).

6.2 Integrated Process Lifecycle Overview

This section intends to give an overview of the mapping analysis results of Annex A in order to have the full set of requirements for an integrated process lifecycle. Table 7 depicts the results in tabular form, where the ISaPro® processes are on the left and the ED-153 objectives on the top. Each green-coloured box represents one mapping of an objective to a process of the ISaPro® process model.

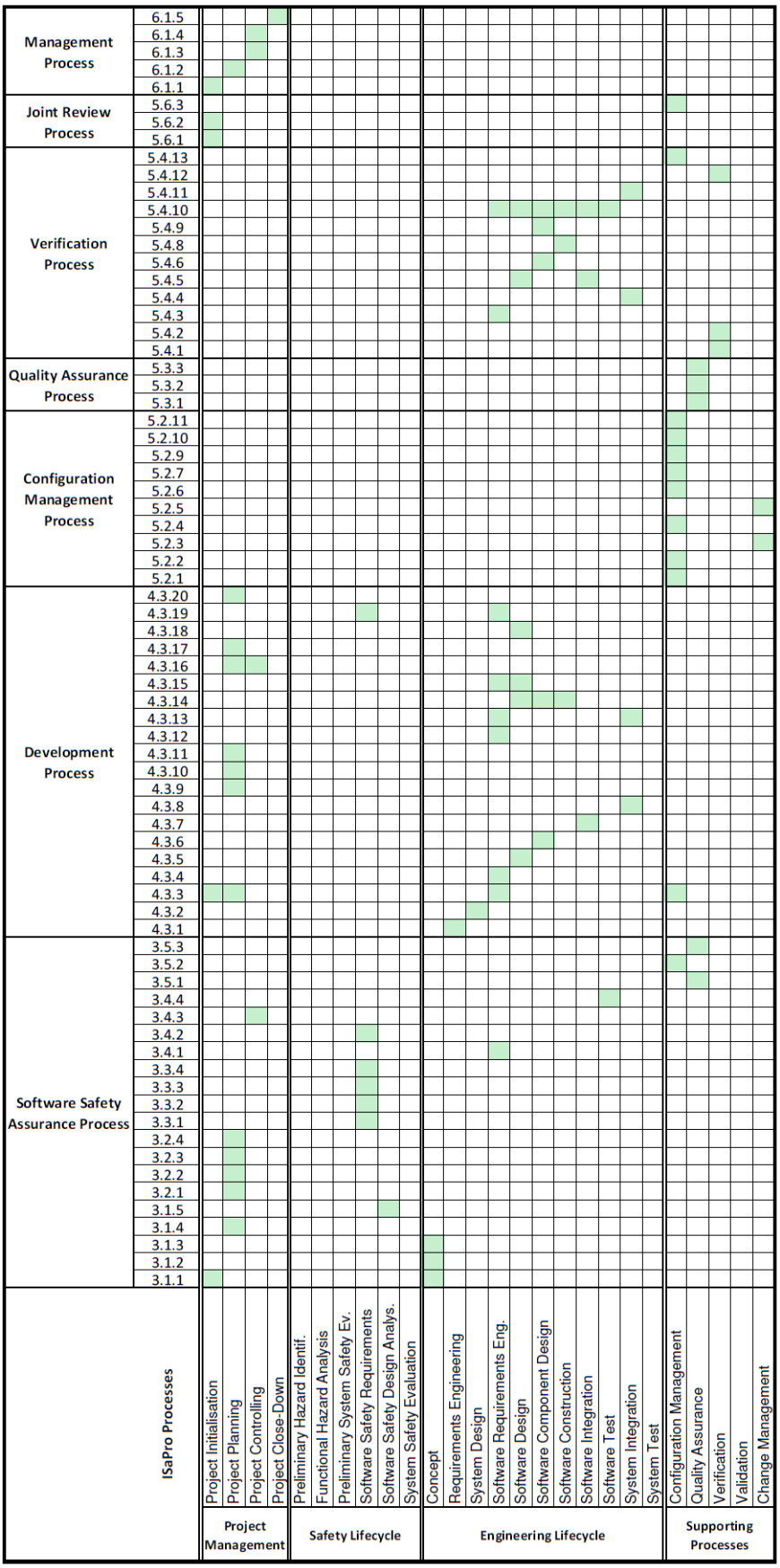

Table 7: Overview of Mappings within Integrated Process Lifecycle ISaPro®

7 Safety Versus Agile Principles

This chapter deals with the comparison of the principles of safety and agility. These principles may either result in synergies (see chapter 7.2) or in conflict with each other (see chapter 7.3). Before these interdependencies can be determined, the basic principles must be identified (see chapter 7.1). The outputs of this analysis highly influence the proposed agile procedure model (see chapter 8).

7.1 Evaluation

The basic agile values and principles are relatively easy to determine due to the common agreement on various approaches in the agile manifesto (see chapter 5.1). Depending on which approach is chosen, the primary focus is set to a different subset of these values. For the comparison in Table 8 some of the very central values were chosen.

In the area of safety there is no such agreed common point of view on values or principles available. There are many different guidelines and standard specifications that deal with safety issues (see chapter 2.3). Each of them concentrates on slightly different topics due to their application in different domains or industries. The following standard specifications have been evaluated for extracting basic safety values and principles, which can then be compared with the agile ones:

- IEC 61508 [IEC10]
- DoD MIL-STD-882E [DoD12]
- EUROCAE ED-153 [EUROCAE09]

One principle which is covered by many safety standards is that of process orientation. This is especially the case in ED-153, where all objectives are mapped to processes of a generic business process framework. In IEC 61508 the safety lifecycle is depicted as a chain of processes and for the purpose of software, the V-model (see chapter 5.5.1) is mentioned. This process orientation should ensure a systematic and sequential approach, which is needed for accomplishing the engineering and safety lifecycle in parallel and stage by stage (as in the integrated process model ISaPro®; see chapter 4).

This process-driven approach has the advantage that the safety analysis can be executed based on the entire definition of requirements and design before starting the implementation. It is necessary to identify all possible hazards and include mitigation strategies in terms of safety requirements into the definition process. This so-called >upfront definition< is the second value, which can be assigned to the term of safety.

Another principle which should be fulfilled by using a systematic process and plan-driven approach is that of evidences. They are of paramount importance to demonstrate to an external authority that the system or software is safe. This is usually done in the form of a safety case (see chapter 2.5).

In the context of the last two principles, upfront definition and evidences, there are two other central principles. The first one is documentation, which mainly should ensure the other principle of traceability. Both are highly supportive of the principle of evidences.

Altogether these previously defined principles contribute to the core principle: the prevention of accidents that could harm humans or property. Basically it is all about this single, but very important principle.

Table 8 provides an overview of the principles of safety and agility evaluated in this chapter. While this is only a rough picture of the two approaches, chapter 7.2 and 7.3 will identify the synergies and conflicts between them.

Safety	Agility
Process Orientation	Individuals and Interactions
Upfront Definition	Evolving Design
Evidences	Simplicity
Documentation	Frequent Delivery
Traceability	Working Software
Prevention of Accidents	Responding to Change

Table 8: Safety versus Agile Principles

After defining and confronting these principles, one basic statement regarding initial specification can be extracted. Whereas agility recommends doing >as much as necessary<, safety endorses doing >as much as possible<. Figure 19 depicts these very different positions in the initial specification phase.

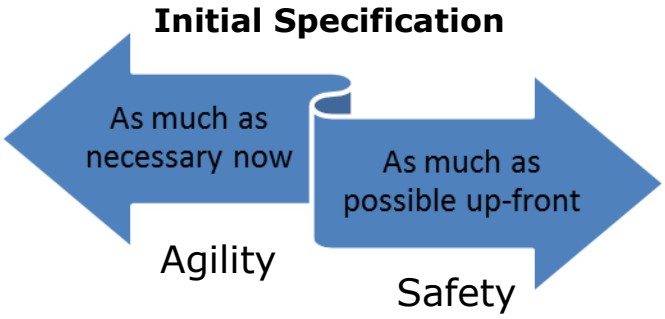

Figure 19: Different Positions on Initial Specification

7.2 Synergies

Although agility and safety seem to be very contrary in some of their attitudes, this section describes some of the potential sources of synergies between the two approaches. In

doing so, this section distinguishes between social factors, process factors and technical practices.

7.2.1 Social Factors

A first social source of synergy is the definition of a team in agile projects. An agile project team should ideally be very interdisciplinary in order to be a cross-functional team [Coh09]. This should avoid frequent handovers of working packages between the different departments involved [Coh09]. Two of the departments which can benefit from this principle are the quality management and safety departments. They can dispatch one of their own experts into the agile project team in order to improve the communication between development and quality/safety experts. For example, these experts can then join the planning meeting for the next iteration to contribute to discussions regarding which topics require further analysis. This kind of collaboration is a lot more interactive than just carrying out quality and safety tasks based on defined requirements.

7.2.2 Process Factors

Within agile approaches it is common to define the term >done<. This so-called “Definition of Done” (DoD) specifies the criteria that must be met in order for an iteration or for a feature/story to be accomplished entirely [Lac12a]. Basically it is a kind of checklist, showing what tasks have to be done to add verifiable value to the potentially shippable software product [Coh09]. For a feature this can be defining test cases, writing code, updating design documents, performing unit tests and many more tasks. Those checklists on various levels should therefore ensure that no necessary tasks are missing after completion. This is in line with the requirements on completeness and integrity in safety-critical environments mandated by standard specifications (e.g. refer to ED-153 objectives 4.3.4 and 5.4.3 in Annex A).

Another enabling process factor is the approach of clarifying technological questions, preferably at short notice. This is ensured with the definition of stories for technological studies that could be done within the first iterations in order to get the results and feedback already at the beginning of the project [WB10]. This should minimise the risks of adopting new technologies without having a detailed initial design phase. The advantage of this kind of rapid prototyping is that new technologies are not evaluated and analysed only within theory but rather by setting up a prototype, which should remediate the lack of clarity. This is also in line with the agile principle of *“fail fast”* [Sho04], which facilitates failing at an early development stage. Such principles can and should be applied to functional and design-related issues as well.

Continuous improvement is a central topic in many agile approaches. It is achieved by using feedback loops on various levels. In Scrum (see chapter 5.3.2) this is done by reflecting each sprint in a meeting at the end of this particular increment. In addition, the frequent delivery of software helps to get fast feedback, which is also an input for improvement. These ideas of continuous improvement are in line with common quality

management systems (e.g. ISO 9001 with its Plan-Do-Check-Act Cycle by Deming [Dem82]), which are required by safety standard specifications as well.

A further central statement of agile approaches is the fact that decisions are made at the latest possible point in time. This is done because of the assumption that late decisions are based on a maximum of information [WB10]. It contributes to a maintainable system due to the fact that simple software solutions are the output of such an approach. There is no need for complex solutions based solely on uncertainty in order to cover a lot of alternatives. Safety-critical systems can profit from easier solutions as well, because the more complexity is added to a software project, the more fault-prone the system is [CB11]. On the other hand late decisions are in contrast to the upfront definition, which is demanded by many safety standard specifications. Therefore this principle is also a potential source of conflict (see chapter 7.3.2).

7.2.3 Technical Practices

One of the technical practices, that of collective code ownership (see chapter 5.2.7), assumes that the source code is consistently structured and well documented. Otherwise it would be impossible to manage the source code of software modules on which the whole team is working. These quality requirements for the code are achieved by working in compliance with agreed coding styles and guidelines. Code reviews and inspections support this compliance by verifying the application of the rules. In addition they ensure the simplicity and maintainability of the software. Such guidelines, reviews and inspections are usually explicitly required by safety standards (e.g. refer to ED-153 objectives 4.3.10 and 5.6.3 in Annex A) as well and therefore in line with the principles of safety.

Probably the largest potential for synergy is in the technical practice of test-driven development (see chapter 5.2.2) including the use of refactoring (see chapter 5.2.3). Refactoring contributes to meeting the requirement for simple and maintainable code. In addition it addresses the sceptical point of view regarding the re-use of software caused by serious accidents in the past (e.g. Therac-25 [LT93]). EUROCAE ED-153, for example, spends a whole chapter on describing requirements and objectives specifically for software which was not developed according to the ED-153 guideline. The most important prerequisite for the refactoring itself is a high test coverage, which therefore implies a wide range of automated unit, system, integration and acceptance tests [FBB+99]. This automatically leads to high code coverage, which is especially important in safety-critical software development. Safety standards like the IEC 61508 or the DO-178B recommend or even require 100 % code coverage [FAA93, IEC10]. Furthermore these standards require tests on every level of the software integration including component, system and acceptance tests.

Working in accordance with the recommendations of test-driven development, where the tests are written before the code, helps the developer to gather immediate feedback about the recent implementation [Kos07]. In addition, with continuous integration the tests automatically answer questions such as whether the recently added code has any impact

on any other software component of the whole platform. By using automated tests a level above as well (e.g. approaches like Acceptance Test Driven Development [Kos07]), unit tests can even provide validation rather than verification only. Therefore constant and especially automated testing is essential for a high quality software product. This is based on the fact that software quality cannot be implemented after the coding phase at the end of the project lifecycle; at this point it can only be verified [Coh09].

Both practices, refactoring and test-driven development, are especially crucial when a software system is developed continuously over time, e.g. a software product delivered in releases over several years. Without refactoring there would be the risk of introducing faults by changing the software, which would lead to more effort in order to correct them again. While these corrections can cause new faults, the critical point is reached when the majority of the development work force is used for corrections instead of implementing new features (also called the “Mythical Man-Month” phenomenon [Bro95]) [Mar08].

7.3 Conflicts

While chapter 7.2 describes the synergies between agility and safety, this chapter analyses the conflicts between them. The potential sources of conflicts arise out of the agile values themselves, from process factors or technical practices.

7.3.1 Agile Values

The first conflicts arise from the agile manifesto (see chapter 5.1.1) itself. The first principle of the manifesto states that individuals and interactions are more valued than processes and tools; this is in contrast to the very process-oriented approaches used in safety-critical development environments. For example, the ISaPro® approach (see chapter 4) even has the term of process in its abbreviation. It consists of a combination of several multidisciplinary processes that ensure the safety of a system. Even the various objectives of the standard specification EUROCAE ED-153 (see chapter 3) are structured in processes. Due to the fact that such process models ensure standardised proceedings, these approaches are widely spread in technological industry. This has positive effects on the likelihood of forgetting a task that might be important for guaranteeing the safety of a software product. But this probability can be decreased by using a definition of done as well (see chapter 7.2.2).

The question is, whether processes and tools will lead to success, even though the involved individuals and their interaction and communication are not in the focus of the project manager. Modern project management approaches acknowledge the increasing importance of individuals. IPMA (International Project Management Association), a federation of various project management associations, has defined a set of competences that a project manager should have in order to successfully manage a project [IPMA06]. A third of these competences are behavioural ones, which deal extensively with communication and interaction within the project team.

The second agile principle, which probably comes into conflict with a safety point of view, is that working software is more valued than comprehensive documentation. Documentation is in line with process-oriented approaches, especially when trying to be compliant with a standard specification. These documents should provide the evidence to a potential assessor that the principles of developing safe software were considered according to the requirements and objectives proposed by the standard specifications. Therefore it is important that documents are written and regularly updated. But of course without any working software, they are useless. According to Cohn [Coh09], documentation should not be neglected, but it should rather concentrate on the most important issues, especially when it is required by standard specifications. Another speciality of agile methodologies is the approach of writing the documentation as the team proceeds with their software product, instead of intensely documenting at the beginning of the project [Lac12a].

Just as documentation is important in the agile world as well as in safety-critical development, working software for a safety-critical application is important too. One of the main drivers of the importance of working software is the increasing trend towards integration of more than one safety-critical software system into a single application [Kni02]. Even the most accurately described interfaces have to be tested in advance in order to be sure that the interworking of multiple systems does not cause any failure. Therefore an early test of the interface is essential where working software is needed. Another potential conflict might be the agile approach of delivering software every iteration while the software product is not finished completely. It is definitely not acceptable to put unfinished and not extensively tested safety-critical software into normal operation. This paradox is mainly caused by the fact that there is the possibility that not all of the safety requirements have been implemented so far and that no comprehensive verification and validation tasks have been done on the software. A possible solution could be to prioritise the safety requirements in a way that they are always implemented first.

In order to relativise the strong statement of not delivering any software that has not been fully verified and validated, it has to be mentioned that in many cases safety-critical systems are tested operationally as well [FHL+98]. This means that the software is deployed in a test environment or in a shadow operation mode (e.g. where a backup system is running in parallel). For this use case of operational tests it is definitely an advantage to have working software frequently delivered. This will increase the assurance of a correctly working software product by using the software as it will be used in the future. In addition to this benefit it allows the development to gather early feedback from an operational point of view, which could be used in order to adapt the software system as it is really needed by the end users. Other possibilities could be that verification and validation are done within the iteration, or in parallel iterations performed by a dedicated test team. While the first possibility requires a high level of automated tests, including validation tests, the second can be accomplished as well using partially manual tests.

The third agile principle values customer collaboration more than contract negotiation. While this statement does not conflict particularly with the safety principles, it is common

in large and complex industrial projects to offer contracts at a fixed price and scope. Such contracts shift the temporal and financial risks from the customer to the contractor. Due to the definition of fixed and strong requirements before starting the project, the customer usually takes the potential risk of the impracticality of the software system. In such contracts agile methods would probably be an advantage for the customer as well. Using an agile understanding, the customer has the opportunity in such a situation to reduce his/her potential risk of impracticality by changing the requirements. Therefore this can lead to win-win-situations as well. Issues that have to be considered when using agile approaches in fixed-bid contracts are also central topics in academic researches by Franklin [Fra08] and Hoda et al. [HNM09].

The definition of customer collaboration within agile approaches includes the availability of the customer, which means that the client is in the position to decide and prioritise in a timely manner [Lac12a]. Some of the approaches even require a representative of the customer, e.g. a business specialist, to participate regularly in local meetings and discussions at the contractors' premises [WB10]. Particularly in international projects these requirements are difficult to meet because of the high costs that this approach generates on the clients' side. A possible solution could be the development of the software project directly on-site, but this vice versa causes costs on the contractors' side. A more practical solution in such cases is the nomination of an internal customer representative who is familiar with the use cases and the needs of the client.

The last agile principle ranks the response to change before following a plan. This statement is closely linked to the one regarding customer collaboration. Within agile development methodologies there is of course a set of requirements designed already in the beginning of the project, but this set is not the final one [WB10]. The requirements in this set can be prioritised, exchanged, or removed and it is even possible to add new requirements, when they get identified. This conflicts with the initial phase of the safety process, where, based on the requirements, the possible hazards of the system are identified and assessed (see chapter 4.3). As change requests and volatile customer needs are part of the majority of development projects including safety-critical ones, such projects and their corresponding safety cases have to respond to change as well.

7.3.2 Process Factors

The fundamental difference between agile and safety-considering approaches is the procedure model. Whereas approaches required by safety standard specifications lean heavily on sequential methodologies like the waterfall- or V-model (see chapter 5.5.1), the agile methodologies are based on an iterative model. This leads to the main problem of where to include those tasks in the agile iterations which were done in the beginning and therefore before the coding in a sequential engineering model. These tasks compromise the definition of all requirements and the development of the design.

According to agile approaches the design has to evolve [Coh09], which implies that the decision concerning the design is made, when it is actually necessary. This decision is therefore taken shortly before the coding for the corresponding requirement is started

based on the maximum of available information [WB10]. This is contrary to safety approaches, where the complete design for the whole software project has to be finished before starting to code. This is necessary in order to verify within the preliminary system safety evaluation (see chapter 4.3.3) and software safety design analysis (see chapter 4.3.5) that all safety requirements are considered in the system and software design and that the design does not cause any harm by itself.

There are also some arguments against an up-front design [PM02]. First of all there are the high costs for this initial phase and secondly the requested changes, which will appear occasionally, are quite expensive because of reworking the initial fixed design [Els07]. This reworking can be done either by going back to the start of the analysis phase in order to complete the specification or by conducting impact assessments which identify and specify the effects on all previously generated process outputs. In contrast to that, there is the agile approach, where plenty small changes have to be done over the whole project cycle. But these can be accomplished relatively cost-efficiently due to the use of test-driven development including refactoring and automated tests [Coh09, Els07].

It is important to mention that an evolving design has some prerequisites. Due to the fact that there will be rework in the code based on changes in the design, it is necessary that the code is well factored (see chapter 5.2.3) and that there is a suite of automated tests (see chapter 5.2.2) in order to detect regression problems at an early stage. [Coh09]

Basically, agile projects have to find a balance between anticipation and adaption [Coh09]. Anticipation reflects the principles of up-front gathering of requirements, design and front-end project planning. In contrast to anticipation, adaption stands for incremental, emerging requirements, design and continuous planning [Hig02]. The appropriate balance for projects is somewhere in between those two extremes. Safety-critical software will very likely be positioned more on the anticipative side in contrast to other software projects. The more the project is based on anticipation, the earlier is there the need for a defined, complete list of system requirements. The disadvantage of this early list is that there are often requirements specified which are skipped or modified during the project’s progress [Coh09].

According to agile literature it is possible to have a dedicated software architect in the team [Coh09, Joh03], although this is actually not intended for an agile team, where every team member is responsible for all the tasks within the scope of software development. The software architect should support the person responsible for managing the prioritisation of the requirements or stories (e.g. the product owner in Scrum) in order to bring in architectural issues and interests. This results in a combination, where the prioritisation is done under the premise of customer and technical needs. Ambler et al. [AL12] calls such an agile software architect an “architecture owner”.

This dedicated role of a software architect fits perfectly with the two goals of an iteration [Coh09]:

- Completion of the planned work of the current iteration
- Preparation for the next iteration

Every single role within an agile team will spend time on achieving both goals, but not in the same proportion [Coh09]. Whereas common team members will spend a bigger part of their time achieving the first goal, other roles like the product manager (or product owner), the agile coach (or Scrum master) or the architect will also spend a significant amount of their time on preparing the following iteration. This, too, is in line with the definition of tasks that a product owner has to achieve within the iterations. According to Lacey [Lac12a], the product owner should spend about 75 % of his/her time on future iterations instead of the current one.

A good example that proves this separation of achieving both goals is an approach which was introduced in the development process of the company Autodesk, Inc., called *"Just-In-Time-Design"* [Sy07]. While the software developers work on finishing the planned work in the current iteration, the interaction designers test the implemented design of the last iteration, preparing the design for the next iteration and gathering customer data for the near future [Sy07].

So a potential solution for the conflicting situation of up-front or emerging design could be that the first iterations are just used to develop an initial design based on rough feature sets. Then this design is evolved over time based on the requirements, which have to be met. In addition, the documentation necessary for the safety assurance of the software, e.g. the safety case (see chapter 2.5), has to be updated regularly in line with the design. This evolution can be supported by a dedicated software architect, who tries to accomplish a conscious design by managing the prioritisation of the requirements. An example for this managing process could be that software parts which were identified by the team as those parts with the most uncertainty are started in one of the first iterations to keep the risk of failure at a low level [Coh09].

7.3.3 Technical Practices

Within agile approaches it is common that requirements are defined as user stories (see chapter 5.2.1). That implies that requirements are examined from an end user's point of view. Derived safety requirements are, in contrast, usually expressed in a very technical way; e.g. *"The system shall not cause hazard 'X' to exist more than 'y' % of the time."* [Fir05]. Agile literature recommends phrasing non-functional requirements in user stories as well, if this is possible and reasonable [Coh08, Dav09]. The issue with non-functional requirements, like safety requirements, is that these are constraints over the whole project lifecycle. This implies that such stories cannot be fully accomplished within a single iteration. A possible solution might be to take such a story into an iteration, accomplish it and put it as a constraint from now on in the definition of done or in the acceptance criteria for each following story [Coh08].

Another issue regarding constraining stories is that in many cases the business value is not visible on the surface. This leads to the threat that those stories get a lower priority due to this lack of clarity. Therefore the corresponding experts, e.g. the safety engineers, have to ensure that those stories are not getting out of sight of the product manager who is responsible for the requirement prioritisation. It might be reasonable to determine a defined quota of story points per iteration for “technical” stories, which have to be accomplished within the iteration [WB10].

Another potential source of danger is the fixed timeframe schedule used in agile approaches. At the beginning of such a time frame the team decides what to accomplish within the iteration. This can lead to time pressure when it is recognised that the work cannot be finished by the end of the iteration. In turn this can cause the decrease of software quality [WB10], which results in short-term success only, because introduced failures lead to more work and the decrease of the team’s performance.

8 Agile Procedure Model

As the previous chapters indicate, an agile procedure model has to be adapted in order to meet the needs of safety-critical software development. It is simply impossible to adopt an agile approach like Scrum or Extreme Programming (see chapter 5.3) without any modifications within such a particular area of software development. This chapter describes the design for an agile procedure model that values the agile manifesto and its principles, while including activities and tasks necessary for safety-critical software development. Due to the inclusion of these activities and tasks, this agile procedure model is called the >Safety Assured Agile Procedure Model<, abbreviated as SAAPM.

> *"[…] it is now widely accepted that [software development] methods should be tailored to the actual needs of the development context."*
>
> *Fitzgerald et al. [FRO03]*

This quote from Fitzgerald et al. [FRO03] points out that the tailoring of software development procedures is widely accepted in industry. However, there is one thing that is of the utmost importance when tailoring a method: there must always be a reason to recommend doing something in other than the preferred way. Therefore the organisation tailoring the method has to be fully aware of the modification and its effects.

The agile procedure model SAAPM relies on the values, principles and practices of agile software development (see chapter 5) while integrating all the necessary tasks that have to be fulfilled in order to be compliant with a safety standard specification (see Annex B). This integration should establish a consensus between the two approaches.

Basically, the procedure model consists of four phases as depicted in Figure 20. The first phase – the pre-game phase – is used for creating the product vision, the initial product backlog and the first high-level software architecture. The aim of the iteration-driven phase is the evolvement of the software architecture based on the backlog items which are implemented. In addition, architectural and safety-related issues are prepared for the upcoming iterations. The third phase – the spin-off phase – is used for the deployment of software. It therefore contains all the preparation tasks necessary for deploying the software into either a test or production environment. The last phase is the wrap-up phase. It takes place when the software system development is finished and comprises the tasks of the spin-off phase supplemented by project close-down activities.

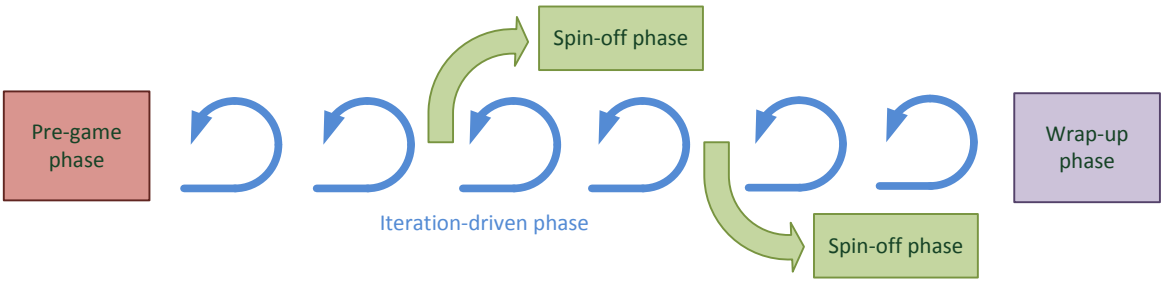

Figure 20: Four Phases of the Agile Procedure Model SAAPM

Chapter 8.1 defines the preconditions and the constraints for such a model. Without a context and constraints the procedure model would have to be very generic, which is not feasible when developing a model for such a specific field of software development. Chapters 8.2 to 8.5 describe the content of each phase of the SAAPM. The central topic in chapter 8.6 is the compliance with EUROCAE ED-153 (see chapter 3). Finally the agile procedure model is evaluated from different perspectives in chapter 8.7.

8.1 Preconditions and Constraints

The first issue that must be resolved before introducing this procedure model is the transition of the organisation itself to an agile approach. Many project teams introducing agile methods in their projects struggle due to the traditional culture of their organisation or lack of adequate support from management [BT05, GBL+04, Hir05]. There is much literature available that documents which prerequisites are necessary within the organisation or which approaches might be successful [CF03]. Therefore this thesis supposes that the organisation has already introduced agile methods or is at least ready to adopt them.

Another issue that deals with the adoption of agile methodologies in organisations is the use of technical practices that are often referenced by agile approaches. As already outlined in chapter 5.2 these technical practices facilitate the achievement of agile values and principles. Therefore this thesis supposes that the agile development model SAAPM includes the responsible use of these technical practices.

A further organisational topic is the size of the project and its teams. Agile approaches fit best with small to medium-sized teams of five to a maximum of nine team members. If a project needs more human resources, teams are divided into multiple of sub-teams. This setup introduces new challenges such as the coordination of overall development and integration, for example. These topics are addressed by some approaches such as “Scrum of Scrums”, which is an approach for scaling agile to larger teams and organisations [Coh09]. For the purpose of this thesis, the SAAPM sticks to small and medium-sized teams, where there is the assumption that only up to three agile teams develop the software.

The difference between the development of a new system and the refinement and enhancement of legacy systems is another issue which has to be considered. As already mentioned in chapter 5.4, agile methods are well suited for “green-field” projects [Kru10], where the system is built up from scratch. The difficulties raised by legacy systems are mainly related to the fact that these systems were often not built considering the principles of agile methodologies. Technical practices such as test-driven development (see chapter 5.2.2) cannot be implemented after the system has been built. The legacy system has to go through an interminable process in which individual challenges appear that have to be met by the development team [BT05]. As there are already some documented experiences of using agile methodologies in combination with legacy systems [Han11, SP04], these are not targeted in this thesis.

The last issue is the safety criticality of the project or product that has to be developed. Safety integrity levels addressing high safety-critical developments require rigorous certification procedures for the software development tools or even the code itself. In the case of EUROCAE ED-153 [EUROCAE09], object no. 4.3.19 requires validation and certification for compilers, linkers and code generation tools when dealing with highly safety-critical systems (see Annex A). As these additional tasks demand tremendous effort, such safety criticality levels are considered as out of scope for this thesis.

8.2 Pre-game Phase

When Schwaber [Sch97] introduced his agile methodology Scrum in 1997, he recommended a preparation phase at the beginning of the software development called “Pre-game”. Although this pre-game phase is not mentioned again in later literature by Schwaber [Sch04], it is considered for this particular approach.

According to Schwaber this initial phase should target planning and architecture. Planning comprises the definition of a software release based on the backlog already known and a rough estimation of its schedule and costs. In the case of a new system or product being developed, this phase contains, as well as analysis, the conceptualisation of the system. The architecture is accomplished by a high-level design for how the items of the product backlog will be implemented. [Sch97]

Schwaber’s definition of the pre-game phase [Sch97] is the origin for the first of four phases representing the particular agile procedure model SAAPM for safety-critical software development. In order to meet the needs of such a development project, this pre-game phase has to be modified and extended. These modifications are introduced in the following paragraphs.

8.2.1 Workshop Organisation

First of all the conditions for this initial phase should be clarified. The phase is carried out in the form of a workshop that lasts several days at a minimum, depending mainly on the size of the prospective software system. It is essential that this workshop does not take

so long that it might be perceived as the entire requirement engineering and design phases of a traditional approach (see chapter 5.5.1). The objective target of this workshop is the effective and efficient rough modelling of the system or product. Therefore the recommended duration of this initial pre-game phase is about three to twenty days depending on the project's context.

The workshop's success relies heavily on the precondition that the skilled and experienced future team members are among the participants. This ensures the interdisciplinary skills needed in order to accomplish this initial phase very efficiently. In the very particular context of safety-critical software development, at least the following roles are required for the workshop:

- Product manager(s)
- Representative(s) of the client(s) or internal customer representative(s) (e.g. product owner)
- Agile coach (e.g. Scrum master)
- Software architect(s) or senior software developer(s) (e.g. architecture owner)
- Safety engineer(s)
- Software developer(s)

The so-called "architecture owner" role is relatively unknown compared to the other roles mentioned [AL12]. It basically represents the software architect in an agile team. While the software architect is the primary creator of the architecture in traditional role perceptions, the architecture owner is primarily responsible for facilitating the architecture modelling. This modelling should be done collaboratively within the development team, where the architecture owner is still the final decision-making authority. [AL12]

The workshop should be well structured in order to proceed efficiently. Therefore it is highly recommended that it is organised into the following three parts:

- Creation of the system or the product vision
- Development of the technical concept
    - Rough catalogue of requirements
    - High-level system architecture
- Performance of the first safety analysis

8.2.2 Part One: Creation of the System or the Product Vision

The first part of the workshop is the creation of a shared system or product vision that inspires all team members. The product owner should take the lead in this part of the workshop due to his/her responsibility for the big picture of the system or product. Basically the aim is to establish a common understanding of how the final system or product should look. Therefore it is essential that all participants understand the requirements set by the customer or the market. In addition, it helps to promote a commonly known, positive side effect that team members who work with a shared vision in mind are more motivated than those who do not [Sch11].

"Imagination gives you the picture. Vision gives you the impulse to make the picture your own."

Robert Collier

8.2.3 Part Two: Development of the Technical Concept

The development of a technical concept including an approximate estimation of costs and lead time is the second part of the pre-game phase. The technical concept is based on the information provided by the customer or the project owner in the form of system requirements and project goals. The technical realisation possibilities that fulfil these goals and requirements are the main content of the technical concept. Therefore it consists of a rough catalogue of requirements and the high-level system architecture. The catalogue of requirements is created using creativity techniques (e.g. brainstorming) performed under the lead of the product owner. In contrast, the high-level system architecture needs a more structured approach, which is guided by the architecture owner.

The development of the key requirements should be achieved in a relatively short period of time. This process is supported by the vision created in the first step, the needs of the customer(s) and market(s) and/or by a formal requirement specification document. The primary goal is to detect the main requirements which are crucial for the functionality of the system or product. In a further step these requirements are written as user stories to an initial product backlog that are ordered according to their business value.

The high-level design of the system architecture includes the definition of the boundaries of the system, its context, its interfaces and finally its internal components. This initial envisioned architecture should ensure a first technical direction and its potential risks that have to be dealt with [AL12]. To be in line with this objective, this modelling of the high-level design should not result in an entire system design. This process should rather consider basic principles linked to agile architecting [Sta11]:

- Attach importance to flexibility and not to inflexible patterns
- As little formalising as possible, but as much as necessary
- Level of documentation should correspond to the particular risk of the system or system component

This modelling phase should also be used for identifying critical components and/or parts of the system. This will help the process of prioritisation in the beginning of the development phase, where the most critical parts should be developed first in order to minimise risk [Lac12a]. To efficiently create such a design that meets all previously targeted objectives, the four-step-approach depicted in Figure 21 can be used:

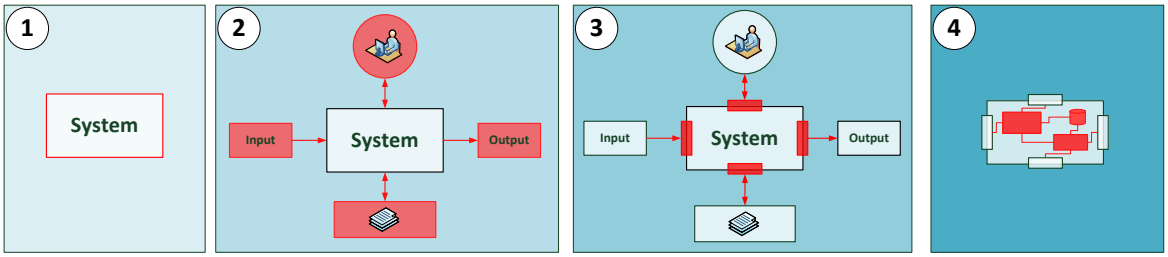

Figure 21: High-level Design of System Architecture

1. Identification of system boundaries

In the first step the boundaries of the system have to be clearly identified. This is necessary in order to determine which parts are inside and outside of the system [SW01]. A clear model of the system including its boundaries also helps to identify hazards in the third part of the workshop (see chapter 8.2.4).

2. Identification of system actors

The next step is the identification of the system actors. Actors are anything that interfaces with the system; e.g. people, hardware, data stores, bus systems, networks or other software. Each of the actors defines a particular role that acts with the system. An entity (e.g. a person) is represented by either one or more actors in the case that the entity takes on different roles with regard to the system. Conversely, several entities can be represented by one actor when they all take the same role. The system itself is perceived as a black box and therefore only its interactions with the world outside are modelled. [Sta11, SW01]

3. Identification of system interfaces

In the third step the interfaces of the system are identified. Interfaces are those parts of the system where the actors interact with the system on its boundary. Such interfaces could be either direct, messaging or user-machine interfaces [Sta11]. It is of the utmost importance that these system interfaces are well specified in order to avoid malfunction of the system.

4. Identification of system's internal components

The last step is the identification of the internal components and their internal interfaces. It helps to identify the building blocks of the system in order to determine their structure and their connections [Sta11]. Furthermore this fourth step helps to get an overview of which components must be built in order to fulfil the initial requirements.

8.2.4 Part Three: Performance of the First Safety Analyses

The system or product vision and its technical concept are the ideal inputs for the first safety analyses, which are necessary in the development of safety-critical software

applications. The advantage of doing these analyses within the workshop in close collaboration with the interdisciplinary team members is that the coverage of identified safety issues is increased. This fact is also emphasised by a statement of Poppendieck et al. [PM02] in which she recommends consulting a lot of knowledgeable people in order to determine all safety issues. In addition, there is the optimal condition that the whole team has basically the same deep system knowledge due to the work that they have achieved in the first two parts of the workshop. This part of the workshop should be guided by the safety engineer, who is also familiar with the methods and tools used in such safety analyses (see chapter 2.4).

This last part of the workshop comprises the first phase of the safety lifecycle, the preliminary hazard identification (see chapter 4.3.1). The main aim of this process is to identify all potential hazards to which the system can lead. Therefore the previously defined interfaces of the system have to be analysed by using checklists, historical data from former projects or creativity methods (e.g. brainstorming).

As this preliminary hazard identification progresses, more detailed analyses can be carried out. These analyses are part of further safety lifecycle processes on the system and software level that have to be started at the latest in this early stage. Those processes are:

- System level: functional hazard evaluation (FHE) and preliminary system safety evaluation (PSSE) (see chapters 4.3.2 and 4.3.3)
- Software level: software safety requirements analysis (SSRA) and software safety design analysis (SSDA) (see chapters 4.3.4 and 4.3.5)

The identification of unacceptable hazards results in the modification and improvement of the high-level design of the system architecture. Afterwards the preliminary hazard identification can be continued. This iterative process lasts as long as the high-level architecture results in hazards that exceed the level of acceptable risk. All safety lifecycle processes which are started in the workshop have to be continued for the duration of the whole project lifecycle in the iteration-driven phase (see chapter 8.3).

Within these analyses different techniques and methods can be used (see chapter 2.4). Apart from their approach, they also differ in their requirements for information [GPM10]. Table 9 summarises the information that is needed in order to perform the particular methods.

Safety Analysis Technique	Required Information
FMEA (see chapter 2.4.2)	• System architecture structure • Failure modes and their effects on components
HAZOP (see chapter 2.4.3)	• System architecture structure • Potential system failure modes (key words) • Failure behaviour of components • Functional component description
FTA (see chapter 2.4.4)	• System architecture structure • System failures (as top events for the fault trees) • Functional component description

Table 9: Required Information per Safety Analysis Technique [based on RCC99, GPM10]

8.2.5 Outputs

After identification of the vision, key requirements, preliminary high-level design and safety-related issues, these results can be aggregated into an initial plan. In addition to these achieved workshop outputs the plan should also include definitions of the further "way of working" [TSS12]:

- Compliance to norms, standards and rules
- Definition of done (see chapter 7.2.2)
- Use of support tools

Agreement on which norms, standards and rules should be fulfilled is necessary to create a common view among all team members. Apart from the safety standards, which are of the utmost importance, design and coding rules should be defined too. Such design and coding guidelines facilitate the team's achieving well-structured and clean source code.

As already described in chapter 7.2.2, the "Definition of Done" [Lac12a] should ensure that each requirement or feature passes all necessary checks in order that it can be ticked off as done. Such a definition should consider at least the following tasks:

- Evaluation of safety impacts on the system
- Implementation (coding) including unit tests
- Update of documentation
- Conducting of code reviews
- Addition of dedicated test cases (for verification and validation)

A wide variety of tools is supported for each particular demand of agile software development. Some of these tools even provide complete solutions in which the aspects of requirements, development and test management are included. Agile practices usually

recommend using simple methods such as index cards and large visible charts [BA04]. While these seem to be adequate for some projects, computer-assisted tools help the team immensely in documenting necessary artefacts and traces which might be required by safety standard specifications or other regulations.

8.3 Iteration-driven Phase

In the second phase of the SAAPM, the software development starts its progress in time-boxed intervals called “iterations” [Sch04]. According to agile definitions (see chapter 5.3.2), these time boxes should last one to four weeks, depending mainly on the circumstances of the project. Because of the vast quantity of influencing factors, Lacey [Lac12a] developed a model in which the iteration length is determined by answering a questionnaire. Due to the availability of this approach, the determination of the iteration length is not considered in this agile procedure model.

8.3.1 Responsibility Assignment

In order to ensure an evolving design and the safety assurance of the software over all upcoming iterations, some of the team members with a particular role have specific tasks. The software developers are mainly working on the core goal of the sprint, the completion of all agreed backlog items. In parallel, a team guided by the product owner including the architecture owner and the safety engineer has further particular responsibilities. This team is referred to as the >Product Architecture Team< in the following paragraphs. Figure 22 depicts the collaboration between these two teams in order to achieve the three main aims of an agile iteration in the SAAPM.

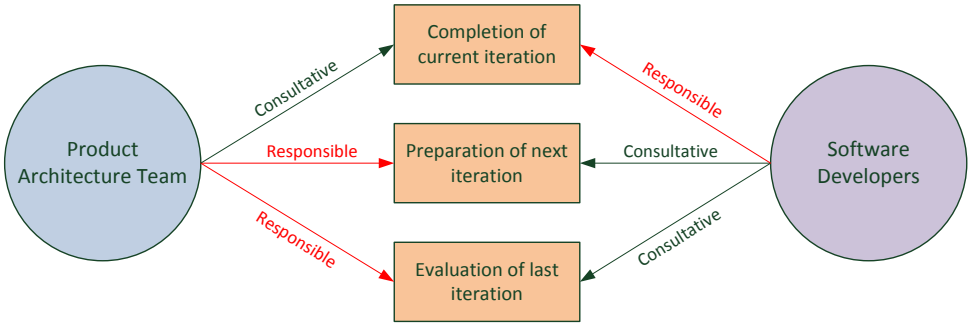

Figure 22: Role Responsibilities

8.3.2 Product Architecture Team

This team is built up of the product owner, the architecture owner and the safety engineer. It has responsibility for the particular tasks of preparing the next iteration and evaluating the last iteration.

Preparation of the next iteration

For the product owner him/herself, the first task – the preparation of the next iteration(s) – is of the utmost importance. As this person is responsible for the product backlog, he/she has to enter new user stories including their acceptance criteria. Furthermore he/she has to prioritise these stories according to their business value. The architecture owner and the safety engineer are basically types of stakeholder. They have to ensure that architectural and safety-related issues are appropriately considered in the product backlog.

The architecture owner is responsible for controlling the system architecture by influencing the order of the product backlog or by entering user stories that are necessary for a well-considered architecture. This architecture has to be prepared for the upcoming requirements in the backlog, especially for the safety-related requirements. The product owner must be convinced that such stories and/or a reordering of the backlog have the business value that justifies the backlog's modification. A similar approach was identified as a practice in some agile projects in India and New Zealand by Hoda et al. [HKN+10]. In their context this approach was called the *"design pipeline"* [HKN+10].

The safety engineer's duty is the continuous safety analysis of the upcoming requirements. It is his/her responsibility to ensure that the impact of these requirements, especially when they are newly entered, is analysed and evaluated. These tasks are the continuation of the software safety requirements and design analyses of the pre-game phase (see chapter 8.2). Therefore it has to be analysed whether these upcoming requirements will introduce new hazards which might affect existing safety requirements or require completely new ones. Afterwards the system architecture has to be assessed to check if it is still capable of dealing with the safety requirements. Tasks that must be achieved by the software developers are inserted as user stories that are put into the product backlog. Their business value is obvious and therefore it is easy to argue for them. In the case that these safety-related requirements are not implemented in the software, the whole system or product is potentially not safety-assured. This may lead to a system that cannot be used in a production environment and therefore is basically useless for the customer, which means that such safety requirements automatically have a high business value for the system.

In summary, all three particular roles must work closely together in order to ensure a well prepared product backlog for the next iteration. This has to be achieved within the iteration and therefore should also be considered as a factor in the determination of the iteration length. To be in line with the agile values and principles (see chapter 5.1) the preparation is done by involving the whole team, establishing commitment to the decisions that have to be made. The result of this process is the timely preparation of the requirements, including their impact on the system architecture and safety assurance, before the next iteration starts. This approach can be called a >Just-in-Time-Safety-Assured-Design<, which is derived from an approach by Sy [Sy07], whereby the user interface design is delivered just in time to the agile development team (see chapter 7.3.2).

Evaluation of the last iteration

Under the premise that the agile team has achieved its goal of the previous iteration and therefore finished all committed backlog items, the evaluation of that iteration can take place.

Therefore the system architecture is critically reviewed by the architecture owner in terms of whether it is applicable for its purpose and the future needs of the system. Necessary changes will be directly integrated in the preparation of the next iteration and will most likely find their way into the product backlog.

The safety engineer verifies that all safety requirements and constraints are well considered and correctly implemented. This should be done in such a way that the remaining risks caused by the system or product are below an acceptable level. Basically the safety engineer performs the tasks of the last safety process within the safety lifecycle: the system safety evaluation (see chapter 4.3.6).

8.3.3 Software Development Team

While the product architecture team has particular tasks, the software development team is mainly working on implementing the user stories that have been committed for this iteration. Apart from this task, the system architecture is refined over the lifecycle of the system according to the preparation done together with the architecture owner.

The first iteration has a special significance, as only the preparation output of the pre-game phase is available. When implementing the previously designed, high-level system architecture, it is recommended that it is done according an approach developed by Cockburn [Coc04] called *"Walking Skeleton"*. This approach is about implementing a tiny architecture that should link together the main architectural sub-elements or components in order to provide minimal end-to-end functionality [Coc04]. This initial walking skeleton can then be refined and extended within the following iterations.

Another important task for the development team is to support the architecture and safety persons in charge. This should be achieved in dedicated meetings in which a particular issue is the topic and the experts discuss it and try to find possible solutions. Such issues can be architectural, safety-related or generic. No matter what kind of issue occurs, the recommendation is that the decision should not be made without seeking advice from the team members.

8.3.4 Documentation

The importance of documentation in safety-critical software development cannot be denied. Whereas sequential approaches document all outputs of the initial analysis and design phases, in agile development this has to be done iteratively. No matter what kind of documentation is produced, it has to be started in the pre-game phase and has to be evolved over the project lifecycle.

The minimum necessary documentation artefacts are:

- Requirements including safety requirements
- Architecture and design documents
- Safety documents (e.g. safety arguments, evidences and safety case)

The safety documentation particularly challenges the safety engineer(s). In order to create a safety case, it is necessary to provide a detailed safety argument that points out how the safety of the system will be achieved. This argument is usually created monolithically as it is designed for the complete software system. Due to the fact that it has to be created in iterations, it is recommended that safety arguments are created per software module [GPM10]. Afterwards these single modular safety arguments can be linked to a software system safety argument that consists of all arguments of its subsystems and an argument that deals with the interactions of the single modules [GPM10]. Such a whole picture is necessary, when the system has to be delivered to go operational (for example in the spin-off or wrap-up phase).

8.3.5 Overall Picture

Figure 23 depicts the overall interaction between the pre-game phase, the iteration-driven phase and the team focusing on their particular tasks.

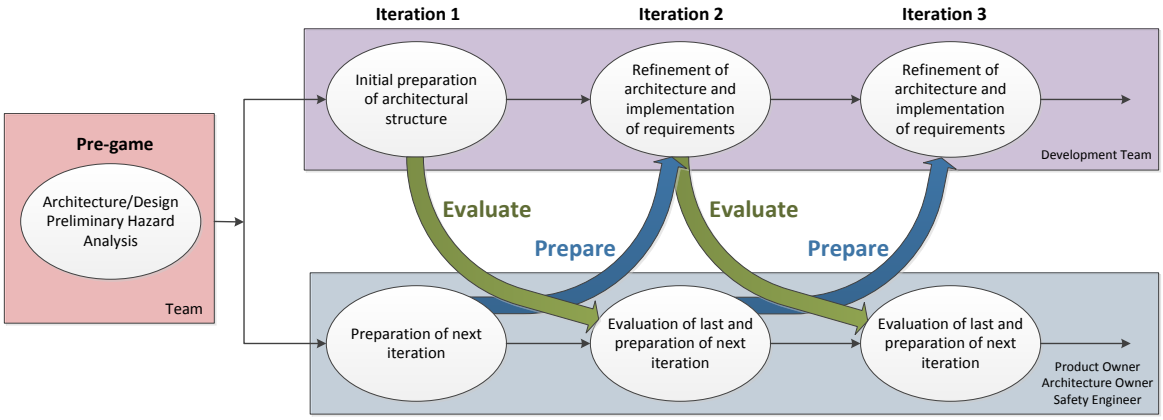

Figure 23: Interaction between Pre-game and Iteration-driven Phase

In the pre-game phase, the whole team works together to achieve a high-level design for the system architecture and the preliminary hazard analysis. When entering the iteration-driven phase, the focus is divided into two sub-groups. That does not imply that the team is split up; it is only the focus which will be different in the upcoming iterations (see also Figure 22). The arrows depicted in Figure 23 show the flow of information for the particular tasks of evaluation and preparation. These flows also depict the interdependencies between the two focused teams. Most of their work depends on the completion of activities of the other team, one iteration before. This fact should indicate that the agile procedure model SAAPM requires rigorous self-discipline and the fulfilment of commitments.

8.4 Spin-off Phase

As depicted in Figure 23, the iteration-driven phase might proceed until the project of developing a system or product is completely finished. However, this differs from the principles of agility, where continuous and frequent delivery is a central principle (see chapter 5.1.2).

For the purpose of the SAAPM, two different deliveries are distinguished. Whereas the first option has the purpose of running the system in a test environment (see chapter 8.4.1), the second has the purpose of taking the system operational (see chapter 8.4.2).

8.4.1 Test System Delivery

In the case of the first delivery alternative (also called *"dry run"* [VB09]), the technical practices recommended for agile approaches should ensure the appropriate functionality of the system. These technical practices are mainly test-driven development (see chapter 5.2.2) and continuous integration (see chapter 5.2.5). Both should make sure that the software is continuously verified by automated unit tests. This should lead to a culture of *"zero defects"*, which was first introduced in 1979 by Crosby [Cro79]. Such a culture is characterised by a team that solves errors as soon as they have been raised by a system or a person. This procedure should ensure that the software is ready for deployment every iteration. Such a test environment can either be the manufacturer's or the customer's. It is highly recommended that this opportunity is used in order to allow the development process to gather immediate and continuous feedback from the testers or even better directly from the customer or end users.

8.4.2 Operational Delivery

The second delivery scenario requires a dedicated phase: the spin-off phase. Within the area of safety-critical development it is often necessary that the system is completely verified and validated before it is ready for operation. In some cases it is necessary to have these tasks performed by an independent party as well. Some business areas even require external approval, e.g. by an authority. As unit tests are not independent [VB09], those tasks have to be done separately. Therefore the current software version is frozen and from that point on treated in a separate path as depicted in Figure 24.

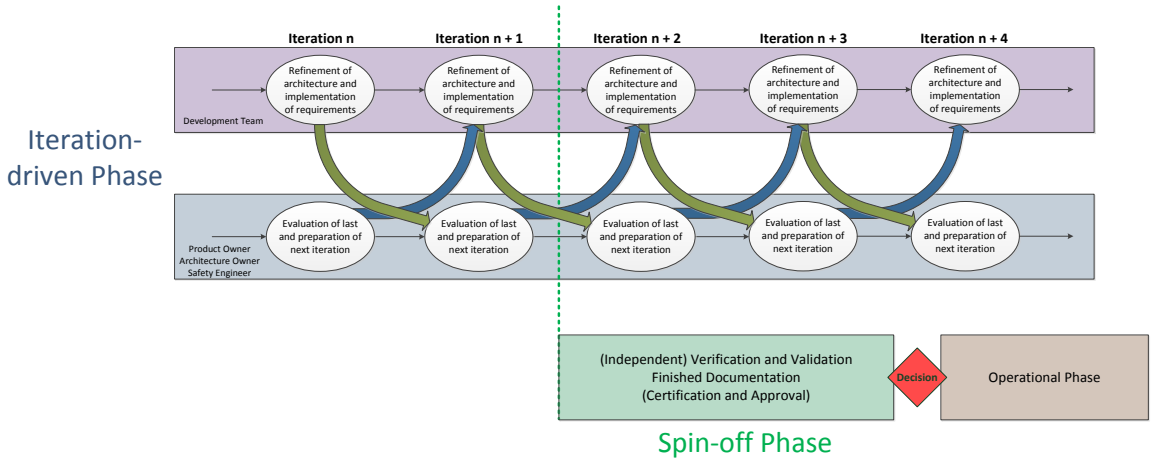

Figure 24: Interaction between Iteration-driven and Spin-off Phase

In this dedicated path the whole preparation for the operational phase is done:

- Comprehensive tests which verify and validate the software
- Completion of the documentation
- Optionally, the certification and approval of the software by an external party

The type of the tests depends on the context of the project and also on the customer. It is recommended that at least integration, load and acceptance tests should be done in this phase. The completion of the documentation, which was iteratively written, is also an important part of this phase. The safety documents in particular should be finished carefully in order to be sure that all necessary issues are covered. After they are finished, they can be released in order to have them officially available. In a last step, the software optionally can be certified and approved, if that is required by law or other regulations. This is usually done by an external authority that approves the correct process of software development within a safety-critical environment. Due to the different tasks that could be necessary, there is no recommendation regarding the lead time for such a spin-off phase.

8.5 Wrap-up Phase

The last phase of the agile procedure model has to achieve basically the same tasks as the spin-off phase (see chapter 8.4). The difference between them is that the wrap-up phase is the phase that finishes the system or product development. Therefore this phase is supplemented by project close-down activities such as the transfer of all the lessons learned into the line organisation. After this phase, the system is handed over to maintenance. Figure 25 depicts the dependencies between the iteration-driven phase and this phase.

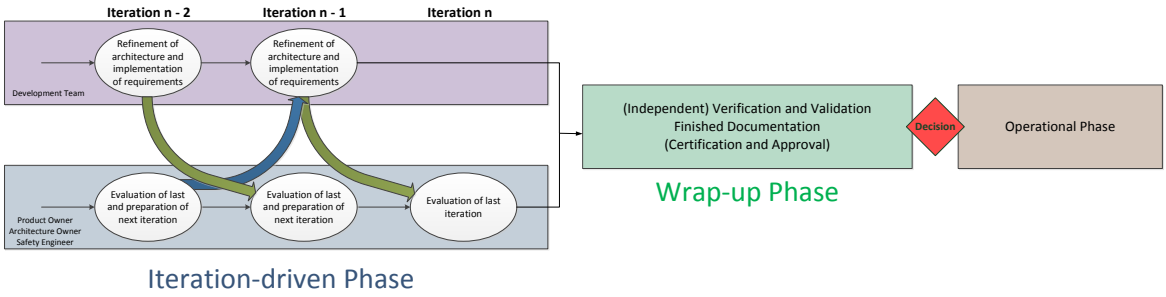

Figure 25: Interactions between Iteration-driven and Wrap-up Phase

8.6 Compliance to Adapted ISaPro® and EUROCAE ED-153

This chapter intends to ensure that the agile procedure model is compliant with the adapted ISaPro® (see Annex B) and therefore with EUROCAE ED-153 (see chapter 3) as well.

This assurance is provided by mapping the activities within the adapted ISaPro® framework, which are necessary to fulfil ED-153 (see Annex B), to the phases of the SAAPM. The mapping is done by analysing in which phase of the SAAPM the activities of the adapted ISaPro® framework fit best. Due to the iterative approach, a major part of the activities have to be carried out in more than a single phase. The detailed results of this process including the data of the evaluation are provided in Annex B.

Basically it is possible to map all activities required by ED-153 to the agile procedure model SAAPM as depicted in Annex B. But in line with the rigorousness of the SWAL (see chapter 3.2), the agile procedure model gets more and more inflated by the required activities. Figure 26 depicts how many activities are required per SWAL per ISaPro® lifecycle. As can be seen, the engineering lifecycle in particular requires many more activities in line with a more rigorous SWAL. Due to the fact that a lot of activities, even those required by SWAL 4, require more granular and detailed activities in the case of more rigorous SWALs, these charts can be viewed only as a high-level perspective.

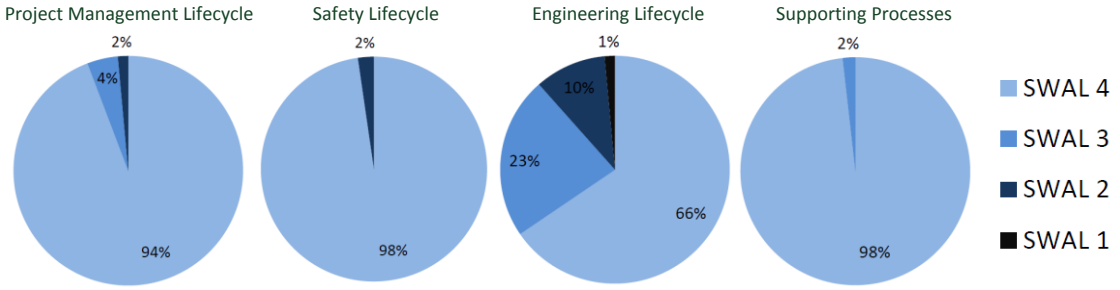

Figure 26: Comparison of Activities required per SWAL in ISaPro® Lifecycles

Figure 27 charts the distribution of all activities in the particular phases of the SAAPM. As a large majority of the activities start in the pre-game phase and then evolve over time in the iteration-driven phase, these two phases have the most activities assigned.

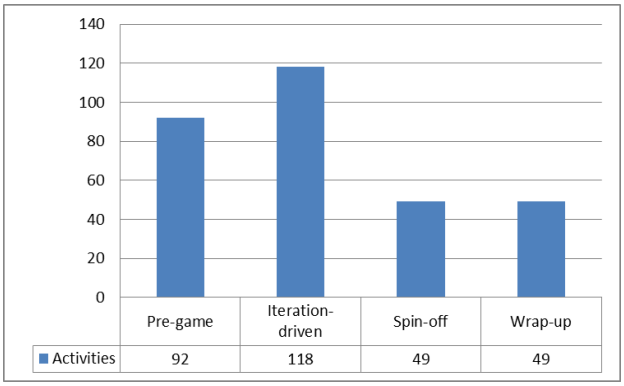

Figure 27: Required Activities per Agile Procedure Model Phase

Therefore it is of utmost importance that the workshop in the pre-game phase is carefully planned and prepared in order to comprise all necessary activities in a reasonable amount of time. In addition, the iterations have to be efficient, as there are more than a hundred tasks that have to be considered. Due to this high number of tasks, agile iterations within the area of safety-critical software are usually longer than in other agile software projects (e.g. [Che09]).

Figure 28 shows the distribution of activities of the ISaPro® lifecycles over the different agile procedure model phases. Whereas the activities of the project management lifecycle are spread evenly over all phases, safety activities are mainly concentrated in the first and second phases. The engineering activities have to be achieved primarily within the iteration-driven phase, while the support processes are evenly distributed throughout the last three phases.

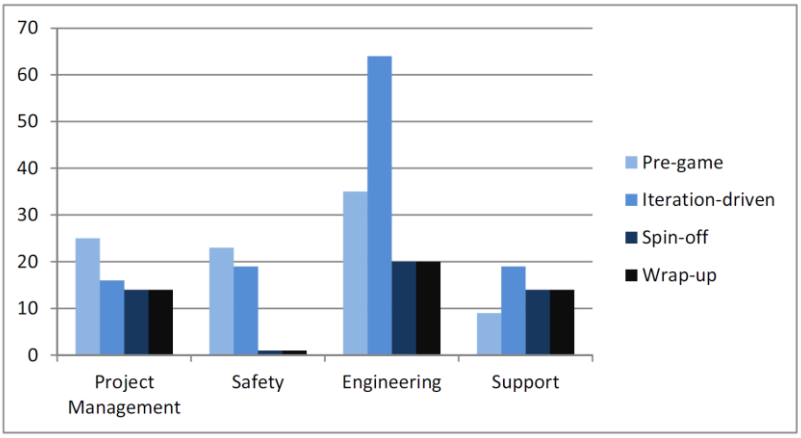

Figure 28: Distribution of ISaPro® Lifecycle Activities over Agile Procedure Model Phases

As outlined already in the previous paragraphs, it is possible to map or assign all activities to the phases of the SAAPM. Nevertheless the team members have to be aware that a development project in the area of safety-critical software systems and applications is more challenging than in many other industries. The achievement of all necessary activities can only be ensured if all team members, including their management, are disciplined enough to efficiently conduct all phases of the model. This should be in line with the agile principles and values as well, according to which the interaction between team members, working software, customer collaboration and the respond to change are crucial.

8.7 Evaluation of Agile Procedure Model

This chapter deals with the final evaluation of the SAAPM. The evaluation consists of the proof of agility, the analysis of advantages and disadvantages and the forecasted applicability of the procedure model.

8.7.1 Proof of Agility

This section discusses the question of whether the agile procedure model is still agile, given that some adaptations have been made, in comparison to common agile approaches. Therefore the agile procedure model SAAPM is evaluated according to the agile values in the form of the agile manifesto (see chapter 5.1.1) [Cun01].

Individuals and interactions *over processes and tools*

In the description of the agile procedure model SAAPM, the term >processes< is used quite often. This is mainly because of the frequent use of this term in safety-relevant domains. In such business areas, everything is organised as and within processes. But of course there is a need for individuals and interactions. Without them, it is impossible to build even a chaotic system.

A process is defined as a network of activities performed by resources that transform inputs into outputs [WS07]. Therefore agile activities can be seen as processes as well. For example, each requirement or user story (the input) is transformed during implementation into a defined functionality (the output). This output is further defined as >done< (see chapter 7.2.2), which ensures that all outputs meet a predefined quality level.

It is essential to choose the appropriate tools. They can make the team members' life a whole lot easier, especially when the project involves multiple and/or globally distributed teams. In addition, the majority of tools ensure documentation and traceability alongside their primary use, which saves a lot of time in providing the necessary artefacts required by safety standard specifications.

Concluding, it can be stated that the agile procedure model SAAPM values individuals and interactions more than processes and tools. Processes and tools should rather be considered as enablers for high-quality and modern software development.

Working software *over comprehensive documentation*

Working software is achieved by the iterative process, where all iterations should be able to deliver a potentially shippable product. In the agile procedure model SAAPM, the so-called spin-off phases handle the delivery of software. The preparation tasks for this deployment depend on whether the system is delivered only to a test system or to a production environment.

Comprehensive documentation is targeted by the agile device that only as much documentation as necessary is created to fulfil the regulations. Within safety-critical software development, documentation cannot be avoided due to the need to provide justification that all safety issues have been considered. Therefore the tools should be used in an intelligent way whereby the automatic mechanisms for documentation and traceability are leveraged.

"***Customer collaboration*** *over contract negotiation"*

Customer collaboration is not directly considered within the SAAPM. By enforcing iterative development and frequent delivery, continuous feedback should be facilitated. Whether this feedback is provided by internal stakeholders, customer representatives or the direct customer is not important to the model itself. To be in line with the agile values, the customer should of course be involved as much as possible.

"***Responding to change*** *over following a plan"*

Responding to change is also facilitated by the iterative development procedure. It allows prioritising of the product backlog continuously over time. This should be done with discernment, as this reordering of the backlog must be in line with architectural and safety-related concerns.

Although there is the agile recommendation to value "responding to change" higher than "following a plan" [Cun01], the process of planning is important. Without planning, the agile process would result in chaos just like any other unplanned process. Therefore planning should be achieved by analysing upcoming requirements, forecasting the progress of the team(s) and communicating with stakeholders.

"Plans are of little importance, but planning is essential."

Winston Churchill

8.7.2 Advantages & Disadvantages

This section compares the agile procedure model SAAPM to traditional approaches (see chapter 5.5.1). More precisely, this section points out the advantages and disadvantages of the model.

The following advantages could be identified, where some of them are in line with common knowledge of advantages using agile methodologies (e.g. [AL12, PW09]):

- Light-weight initial pre-game phase
  - Use of the efficiency of an interdisciplinary, well-structured workshop
  - Benefit from the fact that only the system that is well-known is modelled and where the risk of change is quite low
- Starting with components and modules that are fraught with risk in order to detect shortcomings of the initially modelled system at an early stage
- Benefit from modern technical practices (see chapter 5.2) that focus on maximising the quality of the source code
- Continuous and prompt feedback after implementation
  - Possibility of modifying the system during the development phase
  - Approaching maximal customer satisfaction due to the delivery of a system that fits the customers' real needs
- Securing a well-designed and safe system by considering architectural and safety-related issues in the process of product backlog prioritisation
- Dedicated phases for providing the opportunity for verification and validation of the system by independent and/or external parties

But as there is no model that has only advantages, the following disadvantages could be identified:

- Risk of changing requirements that affect system architecture or safety requirements in such a way that the system has to be extensively modified
- Need for complete upfront design in order to determine the runtime behaviour of the whole system (for example when dealing with tight real-time requirements)
- Iterative creation of documents might require more effort than initial creation due to frequent rewriting necessitated by changes
- Particular technical expertise is needed over the whole lifecycle instead of in the initial planning and design phase only

8.7.3 Applicability

To answer the question of applicability of the SAAPM is quite difficult. In order to provide significant results, the procedure model has to be compared to traditional models within empirical studies. If resources are not the limiting factor, these studies have to be performed on various different safety-relevant software development projects:

- Small co-located to medium-sized globally dispersed teams
- Safety-relevant to highly safety-critical projects
- Software system and software product development

Apart from the fact that the proof of applicability has to be done within empirical studies, the agile procedure model SAAPM fits best to projects that are within the defined preconditions and constraints (see chapter 8.1). This is also circumstantiated by studies on the “agile sweet spot” performed by Kruchten [Kru04, Kru10] (see chapter 5.4). Apart from the factor of safety-criticality, the defined circumstances for the SAAPM are more or less within this agile sweet spot. Kruchten’s advice is therefore the adaptation to exactly these contexts.

> *“[…] we found valuable to first define the context using our model with several factors, then understand in which dimension(s) the project felt outside of an ideal agile sweet spot, and then in turn drive the adoption and possible adaptation of agile practices to this context […].”*
>
> *Philippe Kruchten [Kru10]*

9 Summary

This thesis deals with the question of whether agile methodologies can be used in safety-critical software development projects as well as in non-safety-critical projects. This question is of importance because of the paradigm change in safety-critical industries whereby regularly changing customer requirements and similar challenges have become daily business. Many organisations in other industries dealing with software development are trying to adopt agile methodologies in order to overcome these existing and upcoming challenges. The central question is whether organisations within the safety-critical industries can leverage on agile approaches as well.

To answer the question in the context of safety, the standard specification EUROCAE ED-153 was chosen in order to determine the activities necessary for ensuring software safety. In order not to reinvent the wheel, the objectives raised by ED-153 were mapped to the integrated process model ISaPro®, which is a generic process model that is tailored to meet the particular needs of safety-critical developments. An interesting fact is that the majority of the objectives of ED-153 do not map to the safety lifecycle of ISaPro®. In fact, these objectives map to processes of all lifecycles, which indicates that the assurance of safety is not the task of the safety expert only. Safety assurance is a team approach, where all interdisciplinary team members have to collaborate in order to develop safe software.

After the necessary activities have been identified, the principles of agility and safety are evaluated. This evaluation facilitates the comparison of the two approaches in order to determine potential sources of synergy and conflict. A large number of synergies indicate that the approaches have more substantial similarities in their attitudes than supposed. In contrast to these positive indicators, there are of course many conflicts between agility and safety as well. Particularly the whole agile manifesto seems to be a conflict in itself, as it values each principle of safety less than the paired one. On closer examination the conflicts turn out to be there, but at least the majority of them can be overcome by tailoring or adapting an agile approach.

Based on the results of the first two steps – the mapping process and the evaluation of the principles – an agile procedure model is created in the last stage. It combines safety and agile values in a holistic approach, while leaving space for the activities that have to be conducted in order to ensure software safety. This agile procedure model consists of four phases that are tailored to the demands of safety assurance. While the focus of the first phase (the pre-game phase) is on the efficient development of an initial concept including a vision, a high-level system architecture and a first safety analysis, the second phase (the iteration-driven phase) comprises the agile sprints. In order to compensate for the lack of upfront design, the software architecture and safety analyses are evolved over time during these iterations. To give the software architect and the safety engineer the ability to steer development in the right direction, architectural and safety-related tasks are performed one iteration ahead. In addition, every iteration includes an evaluation of the previous iteration as well in order to ensure proper design and implementation, considering all safety aspects and issues. To satisfy the needs of independent verification

and validation and/or actual certification by an external party, the last two phases handle the deployment of the software into a test or an operational environment.

Due to the fact that such a generic agile procedure model does not fit for any development project, preconditions and constraints for the optimum applicability were defined. Projects within these constraints are new software systems, developed from scratch by a maximum of three agile teams in a moderately safety-critical environment. In addition it is important that the agile principles and practices are already adopted by the organisation in order to avoid organisational trade-offs.

In order to determine compliance with EUROCAE ED-153, all required activities were mapped to the different phases of the agile procedure model SAAPM. Although this mapping is possible, conducting those activities moderately inflates the agile procedure model, depending on the selected SWAL. Therefore it is important that the initial workshop in the pre-game phase is leveraged in such a way that all team members facilitate a lightweight and efficient building of a solid base. This helps in pushing forward the shared vision of creating valuable software under personal responsibility. While having ensured that the agile procedure model fits to safety-related requirements, the model was evaluated in the context of agility as well. The procedure model still values the left principles of the agile manifesto more than those on the right, although it keeps more focus on the principles on the right than other agile approaches.

The aim of this developed agile procedure model is to facilitate the adoption of agile methodologies in the area of safety-critical software development. As it ensures software safety in parallel with following agile values and principles, the model is suitable for this particular environment. Therefore a further research topic could be an empirical study based on this or a similar agile procedure model. Such a model could be assessed by a comparison to the application of a sequential or plan-driven approach in order to determine which approach is more efficient in developing safety-critical software systems. Another research topic could be the further integration of the agile methods into the integrated safety process model ISaPro®. Such an integration could help to incorporate the advantages of agility in the development process of safety-critical systems in a structured way.

> *“There are two primary choices in life: to accept conditions as they exist, or accept the responsibility for changing them.”*
>
> *Denis Waitley*

Glossary

Air navigation services Umbrella term for air traffic management, communication, navigation services, and meteorological services for air navigation and aeronautical information services [EC05].

Air traffic management Approach with the objective of enabling aircraft operators to meet their planned time schedules without compromising an agreed level of safety [EC05].

Availability *"The ability of an item (under combined aspects of its reliability, maintainability and maintenance support) to perform its required function at a stated instant of time or over a stated period of time"* [BS87].

Certification Process performed by an independent authority that approves a system according to the fulfilment of a standard specification.

Code coverage *"[…] a measurement of how thoroughly the automated tests exercise the production code and its source code statements, branches, and expressions"* [Kos07].

Commercial Off The Shelf (COTS) Any item of supply that is a commercial item sold in substantial quantities in the commercial marketplace [OFR10].

Dependability *"Trustworthiness of a system such that reliance can justifiably be placed on the service it delivers"* [Lap92].

Independence There are varying degrees of independence. This degree *"[…] may range from the same person or different person in the same organisation to a person in a different organisation with varying degrees of separation"* [EUROCAE09].

Lifecycle process Refers to the processes that are needed over the whole lifetime of a system or product along the value chain. These processes start in the planning phase of the development and end with the disposal after the system or product is in operation.

Magic project triangle *"In traditional project management, the objects of consideration of project management are the scope, the schedule, and the costs. The relationships among these objects of consideration are [called] the magic triangle"* [CG06].

Maintainability *"The ability of an item, under stated conditions of use, to be retained in, or restored to, a state in which it can perform its required functions, when maintenance is*

performed under stated conditions and using prescribed procedures and resources" [BS87].

Product backlog *"[…] is a prioritized list of features to be added to a [software] product. Unlike a traditional requirements document, a product backlog is highly dynamic; items are added, removed, and reprioritized each iteration as more is learned about the product, the users, the team, and so on"* [Coh09].

Reliability *"The ability of an item to perform a required function, under given environmental and operational conditions and for a stated period of time"* [ISO95].

Safety argument *"[…] is used to demonstrate how someone can reasonably conclude that a system is acceptably safe from the evidence available"* [KW04].

Safety assurance *"All planned and systematic actions necessary to afford adequate confidence that a product, a service, an organisation or a functional system achieves acceptable or tolerable safety"* [EU05].

Safety requirement *"A risk-mitigation means, defined from the risk-mitigation strategy that achieves a particular safety objective, including organisational, operational, procedural, functional, performance, and interoperability requirements or environment characteristics"* [EUROCAE09].

Software complexity *"[…] refers to the extent to which a system is difficult to comprehend, modify and test, not to the complexity of the task which the system is meant to perform; two systems equivalent in functionality can differ greatly in their software complexity"* [BDZ89].

Software component *"The result of the first level of decomposition of the software architecture, so that requirements, actions, objects, input and output flows can be associated to that software component"* [EUROCAE09]. Furthermore a software component *"[…] can be seen as a building block that can be fitted or connected together with other reusable blocks of software to combine and create a custom software application"* [EUROCAE09].

Stakeholders In terms of a software system, stakeholders may be users, customers, suppliers, developers, businesses [HJD10] or anybody who is interested in the result of the software development process.

System *"A collection of entities (elements, components, models, and so forth) that are organised for a common purpose"* [CBB+10].

System interface *"A boundary across which two systems or elements [depending on the type of interface] meet and interact or communicate with each other"* [CBB+10].

System/software architecture Denotes the high level structure of a system [CBB+10]. For the purposes of this thesis, architecture is seen as the output of the design process.

System/software design *"Activities [...] for determining the structure of a specific information system that fulfils the system requirements"* [Bur10].

Technical concept Provides a rough system design, based on the available information supplied by the customer or the project owner. It depicts the technical realisation, which is a mixture of the customers' needs and their technical solutions. [TSS12]

Traceability In software development this term refers to the *"[…] ability to link system or product requirements back to stakeholders' rationales and forward to corresponding design artefacts, code, and test cases"* [Goe11].

Unit test *"A test that verifies the behaviour of some small part of the overall system. What turns a test into a unit test is that the system under test is a very small subset of the overall system and may be unrecognizable to someone who is not involved in building the software"* [Mes07].

Upfront design This term refers to the procedure of specifying all requirements and the complete design of a system based on the requirements before starting the implementation. This is usually intended by traditional system development approaches (see chapter 5.5.1).

Validation *"Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled"* [EC01].

Verification *"Confirmation by examination of evidence that a product, process or service fulfils specified requirements"* [EC10].

Work product Describes any artefact that is produced by a process. These artefacts can include files, documents, parts of the product, services, processes and specifications. [CMMI10]

Bibliography

[AFSA00] Air Force Safety Agency (Kirtland Air Force Base, New Mexico): *Air Force System Safety Handbook*, 2000.

[AgiAll12] Agile Alliance: *The Alliance*. 2012. http://www.agilealliance.org/the-alliance/ [Accessed on 3rd January 2013].

[AL12] Ambler, S.; Lines, M.: *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise.* Boston: IBM Press, 2012.

[ALR00] Avizienis, A.; Laprie, J.-C.; Randell, B.: *Fundamental Concepts of Dependability.* Proceedings of 3rd IEEE Information Survivability Workshop (ISW-2000) (24th – 26th October 2000), Boston, Massachusetts, USA, p. 7 – 12, 2000.

[ALR+04] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C.: *Basic Concepts and Taxonomy of Dependable and Secure Computing.* IEEE Transactions on Dependable and Secure Computing, Vol. 1, Issue 1, 2004.

[And10] Anderson, D.J.: *Kanban: Successful Evolutionary Change for your Technology Business.* Washington: Blue Hole Press, 2010.

[BA04] Beck, K.; Andres, C.: *Extreme Programming Explained: Embrace Change* (2nd edition). Boston: Addison-Wesley Professional, 2004.

[BDZ89] Banker, R.D.; Datar, S.M.; Zweig, D.: *Software Complexity and Maintainability.* Proceedings of the 10th International Conference on Information (ICIS), Boston, USA, p. 247 – 255, 1989.

[BHI+05] Bozheva, T.; Hulkko, H.; Ihme, T.; Jartti, J.; Salo, O.; Van Baelen, S.; Wils, A.: *Agile in Embedded Software Development: State-of-the-Art Review in Literature in Practice.* Agile ITEA Consortium, Agile Deliverable D.1, Version 1.0 (2005.04.08), 2005.

[BN07] Begel, A.; Nagappan, N.: *Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study.* Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM) (20th – 21st September 2007), Madrid, Spain, p. 255 – 264, 2007.

[Boe79] Boehm, B.: *Guidelines for Verifying and Validating Software Requirements and Design Specifications.* Proceedings of the European Conference on Applied Information Technology of the International Federation for Information Processing (EURO IFIP) 1979 (25th – 28th September 1979), London, UK, p. 711 – 719, 1979.

[Boe02] Boehm, B.: *Get Ready for Agile Methods, with Care.* IEEE Computer, Volume 35, Issue 1, p. 64 – 69, 2002.

[Bro95] Brooks, F.P.: *The Mystical Man-Month: Essays on Software Engineering* (2nd edition). Boston: Addison-Wesley Professional, 1995.

[BS87] British Standards Institution: *Quality Vocabulary – International Terms (BS 4778-1:1987).* 1987.

[BT05] Boehm, B.; Turner, R.: *Management Challenges to Implementing Agile Processes in Traditional Development Organizations.* IEEE Software, Volume 22, Issue 5, p. 30 – 39, 2005.

[Bur10] Burd, S.D.: *Systems Architecture* (6th edition). Boston: Course Technology, 2010.

[BV10] Bozzano, M.; Villafiorita, A.: *Design and Safety Assessment of Critical Systems.* Florida, Taylor & Francis, 2010.

[CB11] Chhillar, U.; Bhasin, S.: *Establishing Relationship between Complexity and Faults for Object-Oriented Software Systems.* International Journal of Computer Science Issues (IJCSI), Vol. 8, Issue 5, No. 2, 2011.

[CBB+10] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Merson, P.; Nord, R.; Stafford, J.: *Documenting Software Architecture – Views and Beyond* (2nd edition). Boston: Addison-Wesley Professional, 2010.

[CF03] Cohn, M.; Ford, D.: *Introducing an Agile Process to an Organization.* IEEE Computer, Volume 36, Issue 6, p. 74 – 78, 2003.

[CG06] Cleland, D.; Gareis, R.: *Global Project Management Handbook: Planning, Organizing, and Controlling International Projects* (2nd edition). New York: McGraw-Hill Professional, 2006.

[Che09] Chenu, E.: *Agility and Lean for Avionics.* Lean, Agile Approach to High-Integrity Software Conference (26th March 2009), Paris, France, 2009 http://manu40k.free.fr/AgilityAndLeanForAvionics1.pdf [Accessed on 24th April 2013].

[CLD99] Coad, P.; de Luca, J.; Lefebvre, E.: *Java Modeling in Color with UML: Enterprise Components and Processes.* New Jersey: Prentice Hall, 1999.

[CMMI10] Software Engineering Institute (SEI): *CMMI® for Development, Version 1.3* (CMMI-DEV, V1.3), 2010.

[Coc00] Cockburn, A.: *Balancing Lightness with Sufficiency.* Cutter IT Journal, Volume 13, Issue 11, p. 26 – 33, 2000.

[Coc06] Cockburn, A.: *Agile Software Development: The Cooperative Game* (2nd edition). Boston: Addison-Wesley Professional, 2006.

[Coh04] Cohn, M.: *User Stories Applied: For Agile Software Development.* Boston: Addison-Wesley Professional, 2004.

[Coh08] Cohn, M.: *Non-functional Requirements as User Stories.* 2008. http://www.mountaingoatsoftware.com/blog/non-functional-requirements-as-user-stories [Accessed on 10th February 2013].

[Coh09] Cohn, M.: *Succeeding with Agile: Software Development Using Scrum* (2nd edition). Boston: Addison-Wesley Professional, 2009.

[Coo08] Cooper, R.G.: *The Stage-Gate Idea-to-Launch Process – Update, What's New and NexGen Systems.* Product Innovation Management Journal, Volume 25, Issue 3, p. 213 – 232, 2008.

[Cro79] Crosby, P.B.: *Quality is Free: The Art of Making Quality Certain.* New York: McGraw-Hill Professional, 1979.

[Cun01] Cunningham, W.: *Manifesto for Agile Software Development.* 2001. http://www.agilemanifesto.org/ [Accessed on 3rd January 2013].

[DAS+07] Dyba, T.; Arisholm, E.; Sjoberg, D.I.K.; Hannay, J.E.; Shull, F.: *Are Two Heads Better than One? On the Effectiveness of Pair Programming.* IEEE Software, Volume 24, Issue 6, p. 12 – 15, 2007.

[Dav09] Davies, R.: *Non-Functional Requirements: Do User Stories Really Help?* DevOpsDays '09 (30th – 31st October 2009), Ghent, Belgium, 2009. http://www.methodsandtools.com/archive/archive.php?id=113 [Accessed on 10th February 2013].

[Dem82] Deming, W.E.: *Out of the Crisis.* Cambridge: Massachusetts Institute of Technology, 1982.

[DMG07] Duvall, P.M.; Matyas, S.; Glover, A.: *Continuous Integration: Improving Software Quality and Reducing Risk.* Boston: Addison-Wesley Professional, 2007.

[DoD12] Department of Defense: *Standard Practice – System Safety (MIL-STD-882E).* 2012.

[Dvo09] Dvorak, D.L.: *NASA Study on Flight Software Complexity.* Final Report (5th March 2009), 2009.

[EC01] EUROCONTROL: *EUROCONTROL Safety Regulatory Requirements (ESARR 4) – Risk Assessment and Mitigation in ATM.* Edition 1.0 (5th April 2001), 2001.

[EC05] EUROCONTROL: *EMOSIA (European Model for ATM Strategic Investment Analysis): Air Navigation Service Provider Model* (EMOSIA II/DOC/3.4). Version 5.1, March 2005, 2005.

[EC10] EUROCONTROL: *EUROCONTROL Safety Regulatory Requirements (ESARR 6) – Software in ATM Functional Systems.* Edition 2.0 (6th May 2010), 2010.

[Els07] Elssamadisy, A.: *Patterns of Agile Practice Adoption: The Technical Cluster.* Toronto: C4Media, 2007.

[Eri05] Ericson II, C.A.: *Hazard Analysis Techniques for System Safety.* New Jersey: John Wiley & Sons, 2005.

[EU04] European Union: *Commission Regulation (EC) No 552/2004 of the European Parliament and of the Council of 10 March 2004 on the interoperability of the European Air Traffic Management network (the interoperability Regulation).* Official Journal of the European Union, 2004.

[EU05] European Union: *Commission Regulation (EC) No 2096/2005 of 20 December 2005 laying down common requirements for the provision of air navigation services.* Official Journal of the European Union, 2005.

[EU08] European Union: *Commission Regulation (EC) No 482/2008 of 30 May 2008 establishing a software safety assurance system to be implemented by air navigation service providers and amending Annex II to Regulation (EC) No 2096/2005.* Official Journal of the European Union, 2008.

[EUROCAE09] The European Organisation for Civil Aviation Equipment: *Guideline for ANS Software Safety Assurance (ED-153)*. 2009.

[FAA93] Federal Aviation Administration (U.S. Department of Transportation): *Advisory Circular: RTCA, Inc., Document RTCA/DO-178B*. 1993.

[FBB+99] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Don Roberts: *Refactoring: Improving the Design of Existing Code.* Amsterdam: Addison-Wesley Longman, 1999.

[FHL+98] Frankl, P.G.; Hamlet, R.G.; Littlewood, B.; Strigini, L.: *Evaluating Testing Methods by Delivered Reliability.* IEEE Transactions on Software Engineering Journal, Volume 24, Issue 8, p. 586 – 601, 1998.

[Fir05] Firesmith, D.: *Engineering Safety-Related Requirements for Software-Intensive Systems* (Tutorial T3). 13th IEEE International Requirements Engineering Conference (29th August – 2nd September 2005), Paris, France, 2005.

[Fra08] Franklin, T.: *Adventures in Agile Contracting: Evolving from Time and Materials to Fixed Price, Fixed Scope Contracts.* Agile Conference 2008 (4th – 8th August 2008), Toronto, Canada, p. 269 – 273, 2008.

[FRO03] Fitzgerald, B.; Russo, N.L.; O'Kane, T.: *Software Development Method Tailoring at Motorola.* Communications of the ACM Magazine, Volume 46, Issue 4, p. 64 – 70, 2003.

[Gar06] Gareis, R.: *Happy Projects!* Vienna: MANZ Verlag, 2006.

[Gar09] Garg, A.: *Agile Software Development.* DRDO Science Spectrum, March 2009, p. 55 – 59, 2009.

[Gar12] Gartshore, R.: *Software Development for Safety-Critical Environments – How safe are you?* Programming Research Webinar (23rd August 2012). 2012.

[GBL+04] Grossman F.; Bergin, J.; Leip, D.; Merritt, S.; Gotel, O.: *One XP Experience: Introducing Agile (XP) Software Development into a Culture that is Willing but not Ready.* Proceedings of 2004 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON) (5th – 7th October 2004), Markham, Ontario, Canada, p. 242 – 254, 2004.

[GH12] Guo, Z.; Hirschmann, C.: *An Integrated Process for Developing Safety-critical Systems using Agile Development Methods.* 7th International Conference on Software Engineering Advances (ICSEA) (18th – 23rd November 2012), Lisbon, Portugal, p. 647 – 649, 2012.

[GM03] Gross, M.; McInnes, K.R.: *Kanban Made Simple: Demystifying and Applying Toyota's Legendary Manufactoring Process.* New York: AMACOM, 2003.

[Goe11] Göknil, A.: *Traceability of Requirements and Software Architecture for Change Management.* Doctoral Dissertation, Centre for Telematics and Information Technology (CTIT), University of Twente, Netherlands, 2011.

[GPM10] Ge, X.; Paige, R.F.; McDermid, J.A.: *An Iterative Approach for Development of Safety-Critical Software and Safety Arguments.* Agile Conference 2010 (9th – 13th August 2010), Orlando, Florida, p. 35 – 43, 2010.

[Han11] Hansen, C.B.K.: *Agile on Huge Banking Mainframe Legacy Systems. Is it possible?* EuroSTAR 2011 Conference (21st - 24th November 2011), Manchester, UK, 2011.

[Hei07] Heimdahl, M.P.E.: *Safety and Software Intensive Systems: Challenges Old and New.* Future of Software Engineering (FOSE) (23rd – 25th May 2007), Minneapolis, USA, p. 137 – 152, 2007.

[Hig02] Highsmith, J.: *Agile Software Development Ecosystems.* Boston: Addison-Wesley Professional, 2002.

[Hir05] Hirsch, M.: *Moving from a Plan Driven Culture to Agile Development.* Proceedings of 27th International Conference on Software Engineering (ICSE) (15th – 21st May 2005), St. Louis, Missouri, USA, 2005.

[HJD10] Hull, E.; Jackson, K.; Dick, J.: *Requirements Engineering* (3rd edition). London: Springer Media, 2010.

[HKN+10] Hoda, R.; Kruchten, P.; Noble, J.; Marshall, S.: *Agility in Context.* Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) (17th – 21st October 2010), Nevada, USA, p. 74 – 88, New York: ACM, 2010.

[HNM09] Hoda, R.; Noble, J.; Marshall, S.: *Negotiating Contracts for Agile Projects: A Practical Perspective.* 10th International XP Conference (25th – 29th May 2009), Sardinia, Italy, p. 186 – 191, 2009.

[Hol06] Holler, R.: *Debunking Myths of Agile Development.* Better Software Magazine, May 2006, 2006.

[HSV+12] Haberl, P.; Spillner, A.; Vosseberg, K.; Winter, M.: *Umfrage 2011: Softwaretest in der Praxis.* Heidelberg: dpunkt Verlag, 2012.

[Hug09] Hugos, M.H.: *Business Agility: Sustainable Prosperity in a Relentlessly Competitive World.* New Jersey: John Wiley & Sons, 2009.

[Hya03] Hyatt, N.: *Guidelines for Process Hazards Analysis, Hazards Identification & Risk Analysis* (1st edition). Ontario: Dyadem Press, 2003.

[ICAO12] International Civil Aviation Organization (ICAO): *2013 – 2028: Global Air Navigation Capacity & Efficiency Plan* (Doc 9750, Draft). 12th Air Navigation Conference (19th – 30th November 2012), Montréal, Canada, 2012.

[IEC10] IEC International Electronic Commission: *IEC 61508, Functional Safety of electric/electronic/programmable electronic safety-related systems, Part 1-7 (International Standard)*. Geneva, Switzerland, 2010.

[Ima86] Imai, M.: *Kaizen: The Key To Japan's Competitive Success*. New York: McGraw-Hill/Irwin, 1986.

[IPMA06] IPMA (International Project Management Association): *ICB – IPMA Competence Baseline Version 3.0*. 2006.

[ISO95] International Organisation for Standardisation (ISO): *Quality Management and Quality Assurance – Vocabulary (ISO 8402:1995).* 1995.

[Joh03] Johnston, A.: *The Role of the Agile Architect.* 2003. http://www.agilearchitect.org/agile/role.htm [Accessed 9th February 2013].

[Kel98] Kelly, T.P.: *Arguing Safety – A Systematic Approach to Managing Safety Cases.* Doctoral Dissertation, Department of Computer Science, University of York, United Kingdom, 1998.

[Kin11] King, R.S.: *The Top 10 Programming Languages.* Online article in IEEE Spectrum (October 2011), 2011. http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages [Accessed 13th December 2012].

[Kni02] Knight, J.C.: *Safety Critical Systems: Challenges and Directions*. Proceedings of the 24rd International Conference on Software Engineering (ICSE) (25th May 2002), p. 547 – 550, 2002.

[Kom12] Komus, A.: *Ergebnisbericht (Langfassung) – Studie: Status Quo Agile Verbreitung und Nutzen agiler Methoden.* BPM-Labor Hochschule Koblenz, Version 1.11 (Juli 2012), 2012.

[Kos07] Koskela, L.: *Test Driven: Practical TDD and Acceptance TDD for Java Developers.* Greenwich: Manning Publications, 2007.

[Kru04] Kruchten, P.: *Scaling Down Projects to meet the Agile Sweet Spot*. IBM developerWorks, Volume 13, August 2004, 2004.

[Kru10] Kruchten, P.: *Contextualizing Agile Software Development. 17th* EuroSPI² Conference (1st – 3rd September 2010), Grenoble, France, 2010.

[KST+84] Kano, N.; Seraku, N.; Takahashi, F.; Tsuji, S.: *Attractive Quality and Must-be Quality.* The Journal of the Japanese Society for Quality Control, Volume 14, Issue 2, p. 39 – 48, 1984.

[KW04] Kelly, T.; Weaver, R.: *The Goal Structuring Notation – A Safety Argument Notation.* Proceedings of the International Conference on Dependable Systems and Networks – Workshops on Assurance Cases (28th June – 1st July 2004), Florence, Italy, 2004.

[Lac12a] Lacey, M.: *The Scrum Field Guide: Practical Advice for your First Year.* Amsterdam: Addison-Wesley Longman, 2012.

[Lac12b] Lacey, M.: *Scrum – The Sprint Cycle.* 2012. http://www.mitchlacey.com/intro-to-agile/scrum/the-sprint-cycle [Accessed on 5th January 2013].

[Lad09] Ladas, C.: *Scrumban – Essays on Kanban Systems for Lean Software Development.* Seattle: Modus Cooperandi Press, 2009.

[Lap92] Laprie, J.-C.: *Dependability: A Unifying Concept for Reliable, Safe, Secure Computing.* Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture-Information Processing (7th – 11th September), Madrid, Spain, p. 585 – 593, Amsterdam: North-Holland Publishing, 1992.

[LBB+02] Lindvall, M.; Basili, V.; Boehm, B.; Costa, P.; Dangle, K.; Shull, F.; Tesoriero, R.; Williams, L.; Zelkowitz, M.: *Empirical Findings in Agile Methods.* Proceedings of the 2nd XP Universe and 1st Agile Universe Conference on Extreme Programming and Agile Methods (4th – 7th August 2002), Chicago, USA, p. 197 – 207, London: Springer, 2002.

[LCF13] Larrucea, X.; Combelles, A.; Favaro, J.: *Safety-Critical Software.* IEEE Software, Volume 30, Issue 3, p. 25 – 27, 2013.

[Lev11] Leveson, N.G.: *Engineering a Safer World: Systems Thinking Applied to Safety.* Cambridge: The MIT Press, 2011.

[LT93] Leveson, N.; Turner, C.S.: *An Investigation of the Therac-25 Accidents.* IEEE Computer Journal, Vol. 26, Issue 7, p. 18 – 41, 1993.

[Mac03] MacCormack, A.: *Agile Software Development: Evidence from the Field.* Agile Development Conference (25th – 28th June 2003), Salt Lake City, USA, 2003.

[Mah08] Mah, M.: *How Agile Projects Measure Up, and what this means to you.* Cutter Consortium Agile Product & Project Management, Vol. 9, No. 9, 2008.

[Mar08] Martin, R.C.: *Clean Code: A Handbook of Agile Software Craftsmanship.* Boston: Prentice Hall, 2008.

[Mes07] Meszaros, G.: *xUnit Test Patterns – Refactoring Test Code.* Amsterdam: Addison-Wesley Longman, 2007.

[MTL10] MTL Instruments: *Availability, Reliability, SIL – What's the difference?* 2010. http://www.mtl-inst.com/images/uploads/datasheets/App_Notes/AN9030.pdf [Accessed on 30th March 2013].

[NASA04] National Aeronautics and Space Administration (NASA). *NASA Software Safety Guidebook (NASA Technical Standard: NASA-GB-8719.13)*. 2004.

[NMM05] Nerur, S.; Mahapatra, R.; Mangalaraj, G.: *Challenges of Migrating to Agile Methodologies.* Communications of the ACM Magazine, Volume 48, Issue 5, p. 72 – 78, 2005.

[OFR10] Office of the Federal Register (OFR) – National Archives and Records Administration: *Federal Acquisition Regulations System (Title 48 of the United States Code of Federal Regulations).* Volume 1, Chapter 1, 2010.

[PM02] Poppendieck, M.; Morsicato, R.: *XP in a Safety-critical Environment.* Cutter IT Journal, Volume 15, Issue 9, p. 12 – 16, 2002.

[PW09] Petersen, K.; Wohlin, C.: *A Comparison of Issues and Advantages in Agile and Incremental Development between State of the Art and an Industrial Case.* Journal of Systems and Software, Volume 82, Issue 9, p. 1479 – 1490, 2009.

[RCC99] Redmill, F.; Chudleigh, M.: Catmur, J.: *System Safety: HAZOP and Software HAZOP.* New Jersey: John Wiley & Sons, 1999.

[Ric08] Rico, D.F.: *What is the ROI of Agile vs. Traditional Methods? An analysis of XP, TDD, Pair Programming, and Scrum (Using Real Options).* TickIT International Journal, Issue 4Q08, 2008.

[Ric12] Richards, M.: *FDD: Doing Agile in a Non-Agile World.* ÜberConf 2012 (19th – 22nd June 2012), Westminster, USA, 2012.

[Roy70] Royce, W.W.: *Managing the Development of Large Software Systems.* Proceedings of IEEE Wescon (1st – 6th August 1970), p. 382 – 338, 1970.

[RSSB93] Rail Safety and Standards Board: *Group Standard GM//TT0040 – Safety of People Working on Traction and Rolling Stock* (Issue 2, Revision A). 1993.

[Sch97] Schwaber, K.: *Scrum Development Process.* OOPSLA Business Object Design and Implementation Workshop, London: Springer, 1997.

[Sch04] Schwaber, K.: *Agile Project Management with Scrum.* Redmond: Microsoft Press, 2004.

[Sch11] Schermerhorn, J.R.: *Exploring Management* (3rd edition). New Jersey: John Wiley & Sons, 2011.

[ScrAll12] Scrum Alliance: *What is Scrum?* 2012. http://scrumalliance.org/pages/what_is_scrum [Accessed on 5th January 2013].

[Sho04] Shore, J.: *Fail Fast.* IEEE Software, Volume 21, Issue 5, p. 21 – 25, 2004.

[Sli06] Sliger, M.: *Bridging the Gap: Agile Projects in the Waterfall Enterprise.* Better Software Journal, July/August 2006, p. 26 – 31, 2006.

[SP04] Stevenson, C.; Pols, A.: *An Agile Approach to a Legacy System.* Extreme Programming and Agile Processes in Software Engineering – Proceedings of 5th International XP Conference (6th – 10th June 2004), Garmisch-Partenkirchen, Germany, p. 123 – 129, 2004.

[SS04] Smith, D.J.; Simpson, K.G.L.: *Functional Safety: A straightforward Guide to applying IEC 61508 and related Standards* (2nd edition). Oxford: Elsevier Butterworth-Heinemann, 2004.

[Sta11] Starke, G.: *Effektive Softwarearchitekturen: Ein praktischer Leitfaden* (5. Auflage). Munich: Carl Hanser Verlag, 2011.

[Sto96] Storey, N.R.: *Safety-critical Computer Systems.* New York: Addison-Wesley Longman, 1996.

[SW01] Schneider, G.; Winters, J.P.: *Applying Use Cases: A Practical Guide* (2nd edition). Boston: Addison-Wesley Professional, 2001.

[SW08] Schedl, G.; Winkelbauer, W.: *Practical Ways of Improving Product Safety in Industry.* Proceedings of the 16th Safety-critical Systems Symposium (5th – 7th February 2008), Bristol, UK, p. 177 – 194, 2008.

[Sy07] Sy, D.: *Adapting Usability Investigations for Agile User-centered Design.* Journal of Usability Studies, Vol. 2, Issue 3, p. 112 – 132, 2007.

[TKH12] Tschürtz, H.; Krebs, P.; Hettlinger, L.: *ISaPro®: A Process Model for Safety Applications*. 19th EuroSPI² Conference (25th – 27th June 2012), Vienna, Austria, 2012.

[TS10] Tschürtz, H., Schedl, G.: *An Integrated Project Management Life Cycle Supporting System Safety.* Proceedings of the 18th Safety-Critical Systems Symposium (9 – 11th February 2010), Bristol, UK, p. 71 – 84, 2010.

[TSS12] Tschürtz, H., Sebron, W., Schauer, W.: *Integrativer Safety Process (ISaPro®).* Vienna Institute for Safety & Systems Engineering, Version 2.2 (16.02.2012), 2012.

[VB09] VanderLeest, S.H.; Buter, A.: *Escape the Waterfall: Agile for Aerospace.* 28th Digital Avionics Systems Conference (DASC) (23rd – 29th October 2009), Orlando, Florida, USA, p. 6.D.3-1 – 6.D.3-16, 2009.

[VerOne13] VersionOne: *State of Agile Survey 2012 (7th Annual).* 2013. http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf [Accessed on 13th April 2013].

[Vin06] Vincoli, J.W.: *Basic Guide to System Safety* (2nd edition). New Jersey: John Wiley & Sons, 2006.

[Vuo11] Vuori, M.: *Agile Development of Safety-Critical Software.* Tempere University of Technology, Department of Software Systems, Report 14, 2009.

[Wal04] Wallmüller, E.: *Risikomanagement für IT- und Software-Projekte: Ein Leitfaden für die Umsetzung*. Munich: Hanser Verlag, 2004.

[WB10] Wolf, H.; Bleek, W.-G.: *Agile Softwareentwicklung: Werte, Konzepte und Methoden* (2. Auflage). Heidelberg: dpunkt Verlag, 2010.

[Wel02] Welch, N.T.: *What is System Safety?* System Safety: A Science and Technology Primer by The New England Chapter of the System Safety Society (Revision A), 2002.

[Wes12] West, D.: *Lean ALM – Managing Flow rather than Disciplines.* EclipseCon 2012 Conference (26th – 29th March 2012), Reston, Virginia, 2012.

[WKM97] Wilson, S.P.; Kelly, T.P.; McDermid, J.A.: *Safety Case Development: Current Practice, Future Prospects.* 12th Annual CSR Workshop on Safety and Reliability of Software Based Systems (12th – 15th September 1995), Bruges, France, p. 135 – 156. London: Springer, 1997.

[WS07] Wisner, J.D.; Stanley, L.L.: *Process Management: Creating Value Along the Supply Chain.* Ohio: Thomas South-Western, 2007.

[York11] University of York: *GSN Community Standard Version 1 (November 2011).* 2011.

[Zem08] Zemrowski, K.M.: *Impacts of Increasing Reliance on Automation in Air Traffic Control Systems.* 2nd Annual IEEE Systems Conference 2008 (7th – 10th April 2008), p. 1 – 6, 2008.

List of Figures

List of Tables

Annex A: EUROCAE ED-153 Mapping Tables

This appendix includes detailed mapping of the EUROCAE ED-153 [EUROCAE09] objectives to processes of the ISaPro® framework. For an overview of the objectives, which are considered in the scope of the thesis, please see chapters 3.3 and 3.4. The ISaPro® framework is described in detail in chapter 3.

Therefore each activity of the ED-153 objectives (which is a sub-objective) is mapped to the most suitable process of the ISaPro®. At the end of each chapter, there will be a compact summary from the perspective of the particular ISaPro® processes in the form of tables. Apart from the mapping information, these tables might contain comments on why the mapping was done and if the ISaPro® process therefore has to be extended. The second case might be necessary if there was no activity within the ISaPro® framework that fulfils this particular objective of ED-153.

The required detailed activities per each process, including those that were added especially for ED-153, are available in Annex B.

Legend

Each objective of the EUROCAE ED-153 [EUROCAE09] standard is listed in the following form:

Objective:	**[N°]**	**[Name of Objective]**	**[SWAL]**	**[C/L]**
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
[Activity N°]	[Description of Activities]		[ISaPro® Process]	

[N°]	Number of the objective in EUROCAE ED-153 standard [EUROCAE09].
[SWAL]	Objective has to be satisfied by listed and higher SWAL levels. If there is more than one level listed, then each activity within the objective lists its required SWAL level on its own. For further general information on SWAL levels, see chapter 3.2.
[C/L]	Indicates whether the software supplier has the lead for taking responsibility [L] or must only contribute [C] (see chapter 3.3).
[Activity N°]	Number of the activity, which is a sub item of an objective, in EUROCAE ED-153 standard [EUROCAE09].
[Description of Activities]	Description of each activity in EUROCAE ED-153 standard [EUROCAE09].
[ISaPro® Process]	Result of the analysis on how the achievable activity can be mapped on a defined integrative process model, in that case, the ISaPro®. For detailed information on the process model, including processes, see chapter 4; for detailed information on activities per process, see Annex B.

Software Safety Assurance System

Software Safety Assessment Initiation

Objective:	3.1.1	System Description	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.1.1.1	The Software purpose shall be defined.		Project Initialisation	
3.1.1.2	Operational scenarios shall be defined (e.g. HMI).		Project Initialisation	
3.1.1.3	The Software and System functions and their relationships shall be defined.		Concept	
3.1.1.4	Software boundaries shall be defined (e.g. operational, time).		Concept	
3.1.1.5	Software external interfaces shall be described.		Concept	

Objective:	3.1.2	Operational Environment	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.1.2.1	The Software and its environment (physical, operational, control functions, legislative etc) shall be described in sufficient detail to enable the safety lifecycle tasks to be satisfactorily carried out.		Concept	

Objective:	3.1.3	Regulatory Framework	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.1.3.1	Applicable safety regulatory objectives and requirements shall be identified.		Concept	

Objective:	3.1.4	Applicable Processes and Guidance	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.1.4.1	Processes and Guidance applicable to the Software Assurance shall be agreed.		Project Planning	

Objective:	3.1.5	Risk Assessment and Mitigation Process Output	SWAL 4	C
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.1.5.1	The system level risk assessment and mitigation identification shall be reassessed at the software level to ensure it is consistent with the software architecture/design.		Software Safety Design Analysis	

Software Safety Assessment Planning

Objective:	3.2.1	Software Safety Assessment Approach	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.2.1.1	The overall approach for the Software Safety Assessment across Software Lifecycle shall be defined.		Project Planning	

Objective:	3.2.2	Software Safety Assessment Plan	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.2.2.1	A plan describing the software safety assessment steps shall be produced (e.g. approach, relations between safety assessment and software lifecycle, deliverables (content and date of delivery), relations with software/system major milestones, project risk management due to safety issues, responsibilities, persons, organisations, risk classification scheme, safety objectives definition approach, hazard identification methods, safety assurance activities, schedule, resource).		Project Planning	

Objective:	3.2.3	Software Safety Assessment Plan Review	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.2.3.1	The Software Safety Assessment plan shall be reviewed and commented for approval by NSA.		Project Planning	

Objective:	3.2.4	Software Safety Assessment Plan Dissemination	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.2.4.1	The Software Safety Assessment plan shall be disseminated to the impacted parties.		Project Planning	

Software Safety Requirements Specification

Objective:	3.3.1	Failure Identification	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.3.1.1	Potential failures shall be identified by considering various ways Software can fail and by considering the sequence of events that lead to the occurrence of the failure.		Software Safety Requirements Analysis	
3.3.1.2	A list of single, consequential and common modes of failure shall be drawn up.		Software Safety Requirements Analysis	

Objective:	3.3.2	Failure Effects	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.3.2.1	The effects of failure occurrence shall be evaluated.		Software Safety Requirements Analysis	
3.3.2.2	The hazards associated with software failure occurrences shall be identified in order to further complete the list of hazards initiated during Risk Assessment and Mitigation process (e.g. FHA and further completed during PSSA).		Software Safety Requirements Analysis	

Objective:	3.3.3	Assessment of Risk	SWAL 4	C
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.3.3.1	The initial Risk Assessment and Mitigation process (e.g. FHA and further completed during PSSA) shall be revisited based upon the outcome of 3.3.1 and 3.3.2.		Software Safety Requirements Analysis	

Objective:	3.3.4	Software Requirements Setting	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.3.4.1	Software Requirements shall be compliant with the Safety Objectives to which the Software contributes and System Safety Requirements. *Note: The definition of "compliant" has to be developed as part of the argument sustaining the demonstration of this objective. This definition should include the traceability with the above level of requirements, the demonstration of the necessity, sufficiency, appropriateness and relevance of the requirements to satisfy the above level of requirements.*		Software Safety Requirements Analysis	

Software Safety Assessment Validation, Verification and Process Assurance

Objective:	3.4.1	Software Safety Assessment Validation	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.4.1.1	The Software Safety Assurance System shall provide an approach to justify that Software Requirements are complete and correct.		Software Requirements Engineering	

Objective:	3.4.2	Software Safety Assessment Verification	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.4.2.1	The software Requirements shall be consistent with functions to mitigate the effects of the hazard and the Safety Objective of the hazards.		Software Safety Requirements Analysis	

Objective:	3.4.3	Software Safety Assessment Process Assurance	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.4.3.1	The software Safety Assessment shall be performed completely. *Note: In accordance with the approved SW safety plan, in conformance with ANSP Safety Management System and in compliance with applicable safety regulatory requirements.*		Project-Controlling	

Objective:	3.4.4	Software Safety Assurance	SWAL 4	L
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.4.4.1	Demonstration and Assurance that SW requirements are satisfied shall be provided.		Software Test	

Software Safety Assessment Completion

Objective:	**3.5.1**	**Document Software Safety Assessment Process Results**	**SWAL 4**	**L**
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.5.1.1	The Software Safety Assessment process results shall be documented.		Quality Assurance	

Objective:	**3.5.2**	**Software Safety Assessment Documentation Configuration Management**	**SWAL 4**	**L**
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.5.2.1	Software Safety Assessment documentation shall be put under configuration management.		Configuration Management	

Objective:	**3.5.3**	**Software Safety Assessment Documentation Dissemination**	**SWAL 4**	**L**
Activities [EUROCAE09]:			**ISaPro® Mapping:**	
3.5.3.1	Software Safety Assessment documentation shall be disseminated to impacted parties. *Note: This document does not presume who the impacted parties are. They are defined in accordance with the approved SW safety plan, in conformance with ANSP Safety Management System and in compliance with applicable safety regulatory requirements.*		Quality Assurance	

Summary

Table 10 shows the compacted results of the mapping process for the objectives of the software safety assurance system. The table contains only those ISaPro® processes, which have at least a single mapping to one of the safety assurance system objectives.

ISaPro® Processes [TSS12]	**ED-153 Objective N° [EUROCAE09]**	**Comments**
Project Management Lifecycle		
Project Initialisation	3.1.1	
Project Planning	3.1.4, 3.2.1, 3.2.2, 3.2.3, 3.2.4	3.2.1 / 3.2.2 / 3.2.3 / 3.2.4: Assumption that safety planning is part of the project planning process. It will be necessary to involve safety experts (e.g. safety engineer).
Project Controlling	3.4.3	3.4.3: Particular attention on complete performance of software safety assessment.
Safety Lifecycle		
Software Safety Requirements Analysis	3.3.1, 3.3.2, 3.3.3, 3.3.4, 3.4.2	3.3.1 / 3.3.2: Strong focus on software failures, therefore mapped to software safety process. 3.3.3: Added activity for updating initial risk assessment and mitigation process.
Software Safety Design Analysis	3.1.5	3.1.5: Added activity for ensuring consistency to PSSE.

Engineering Lifecycle		
Concept	3.1.1, 3.1.2, 3.1.3	
Software Requirements Engineering	3.4.1	
Software Test	3.4.4	
Support Processes		
Configuration Management	3.5.2	
Quality Assurance	3.5.1, 3.5.3	3.5.1 / 3.5.3: Assumption that the documentation of the software safety assurance results and their dissemination are part of the quality assurance process. Therefore own independent activities are created.

Table 10: Mapping of Software Safety Assurance System Objectives

Primary Lifecycle Processes

Development Process

Objective:	**4.3.1**	**System Requirements Analysis**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.1.1	The system requirements specification shall describe, as a minimum: • functions and capabilities of the system; • business/performance, organisational and user requirements; • safety, security, human-factors engineering (ergonomics), interface, operations, and maintenance requirements; design constraints and validation requirements.		Requirements Engineering

Objective:	**4.3.2**	**System Architectural Design**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.2.1	System requirements shall be allocated among hardware, software, people and procedures.		System Design

Objective:	**4.3.3**	**Process Implementation**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.3.1	A software lifecycle model appropriate to the scope, magnitude, and complexity of the project shall be defined and placed under configuration management.		Project Planning, Configuration Management
4.3.3.2	It shall include, as a minimum: • end of activity/phase criteria for each activity/phase • joint technical review for each activity/phase		Project Initialisation
4.3.3.3	Standards/Rules, methods, tools, and computer programming languages shall be selected, tailored and used according to the SWAL.		Software Requirements Engineering
Note: Process implementation includes lifecycle definition, output documentation, output configuration management, SW products problems, environment definition, development plan, COTS			

Objective:	**4.3.4**	**Software Requirements Analysis**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.4.1	The developer shall establish and document software requirements, using software requirements standards/rules as defined per Objectives 4.3.9 & 4.3.10.		Software Requirements Engineering
4.3.4.2	The Software Requirements shall (*4.3.4.2*), as a minimum: • specify the functional behaviour of the ANS software, capacity, accuracy, timing performances, software resource usage on the target hardware, robustness to abnormal operating conditions, overload tolerance. • be complete and correct; • comply with the System Requirements; • an identification of the configuration/adaptation data range.		Software Requirements Engineering
4.3.4.3	Algorithms shall be specified.		Software Requirements Engineering

Objective:	**4.3.5**	**Software Architectural Design**	**SWAL 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.5.1	The developer shall transform the requirements for the software into an architecture that describes its top-level structure and identifies the software components. *Note: The scope of this objective is top level SW architecture definition, top level interfaces design, SW integration definition, SW architecture definition criteria*		Software Design

Objective:	**4.3.6**	**Software Detailed Design**	**SWAL 2**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.6.1	The developer shall develop a detailed design for each software component of the software using software design standards/rules. *Note: The scope of this objective is SW detailed design definition, interfaces design, SW Units tests definition.*		Software Component Design

Objective:	**4.3.7**	**Software Integration**	**SWAL 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.7.1	An integration plan shall be developed to integrate the software units and software components into the software.		Software Integration
4.3.7.2	The plan shall include verification/test requirements, procedures, data responsibilities, and schedule.		Software Integration
4.3.7.3	The plan shall be documented.		Software Integration
Note: The scope of this objective is SW integration plan, SW integration definition, user documentation, SW validation preparation, SW integration evaluation (partially).			

Objective:	**4.3.8**	**Software Installation**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.8.1	A plan shall be developed to install the software product in the target environment as designated in the contract.		System Integration
4.3.8.2	The resources and information necessary to install the software product shall be documented and made available before installation.		System Integration

Objective:	**4.3.9**	**Standards/Rules Definition – Development Plan**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.9.1	The developer shall develop plans for conducting the activities of the development process.		Project Planning
4.3.9.2	The plans shall include as a minimum: specific standards/rules, methods, tools, actions and responsibility associated with the development and validation of all requirements including safety. If necessary, separate plans may be developed.		Project Planning
4.3.9.3	These plans shall be documented and executed.		Project Planning

Objective:	**4.3.10**	**Standards/Rules – Software Development Plan**	**SWAL 2 – 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.10.1	The developer shall identify SW Requirements standards/rules (note minimum content identified in objective 4.3.4)$_{SWAL4}$.		Project Planning
4.3.10.2	The developer shall identify SW Design Standards/Rules$_{SWAL3}$.		Project Planning
4.3.10.3	The developer shall identify SW Coding Standards/Rules$_{SWAL2}$.		Project Planning
Also, references to the standards/rules for previously developed software, including COTS software, if those standards/rules are different.			

Objective:	**4.3.11**	**Requirements Development Management – Software Development Environment**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.11.1	The developer shall identify the selected software development environment in terms of: (1) The chosen requirements development method(s), procedure(s) and tools (if any) to be used. (2) The hardware platforms for the tools (if any) to be used *Example: Method(s) are for example: SADT, SART, OOD…, though procedures are organisational ways of performing requirement management.*		Project Planning

Objective:	**4.3.12**	**Use of a Requirement Specification Tool**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.12.1	A Requirement specification tool shall be used.		Software Requirements Engineering

Objective:	**4.3.13**	**Resource Management**	**SWAL 3 – 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.13.1	A necessary margin with regards usage of resources (e.g. memory, CPU load, drivers, …) for safety purpose shall be specified$_{SWAL4}$.		Software Requirements Engineering
4.3.13.2	The margin shall be measured or verified to ensure satisfaction of the specification$_{SWAL3}$.		Software Test
4.3.13.3	If many software share the same resources, then the margin shall be evaluated at system level$_{SWAL3}$.		System Integration

Objective:	**4.3.14**	**Rationale for Design Choices especially Real Time Oriented One**	**SWAL 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.14.1	The developer shall define real-time design features of software components at architectural design level. A set of properties, such as the following, shall be identified: • tasks and run- time aspects (priority, events, communications, ….) • interruptions (priorities, delay management, SW watchdog…) • treatment & propagation of errors (detection & recovering mechanisms, ….) • data management (protection & deadlock mechanisms, ….) • initialisation/ stop (exchange of data during these phases)		Software Design

Objective:	**4.3.15**	**Traceability**	**SWAL 1 – 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.15.1	The developer shall ensure there is traceability between System and Software requirements$_{SWAL4}$.		Software Requirements Engineering
4.3.15.2	The developer shall ensure there is traceability between Software requirements and Software design (Software component level, architectural design)$_{SWAL3}$.		Software Design, Software Component Design
4.3.15.3	The developer shall ensure there is traceability between Software Architectural Design and Code$_{SWAL2}$.		Software Construction
4.3.15.4	The developer shall ensure there is traceability between Code and Executable$_{SWAL1}$.		Software Construction

Objective:	**4.3.16**	**Traceability – Verification/Transition Criteria**	**SWAL 2 – 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.16.1	The developer shall describe the software lifecycle processes to be used to form the specific software lifecycle(s) to be used on the project, including the transition criteria for the software development processes$_{SWAL3}$.		Project Planning
4.3.16.2	All essential information from a phase of the software lifecycle needed for the correct execution of the next phase shall be available and verified$_{SWAL3}$. *See also evaluation criteria for Specification, design, code, test, integration.*		Project Controlling
4.3.16.3	Transition criteria for all phases shall be defined$_{SWAL2}$.		Project Planning
4.3.16.4	Transition criteria for Requirements Analysis and Verification phases shall be defined$_{SWAL3}$.		Project Planning

Objective:	4.3.17	Design Tool – Software Development Environment	SWAL 3
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.17.1	If a design tool is used, then the developer shall identify the selected software development environment in terms of: (1) The chosen design method(s), procedure(s) and tools (if any) to be used. (2) The hardware platforms for the tools (if any) to be used.		Project Planning

Objective:	4.3.18	Use of Design Tool	SWAL 3
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.18.1	A design tool shall be used.		Software Design

Objective:	4.3.19	Code Generation Environment	SWAL 1 – 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.19.1	**Software Development Environment:** The developer shall identify the selected software development environment in terms of$_{SWAL4}$: (1) The programming language(s), coding tools, compilers, linkage editors and loaders to be used, (2) The hardware platforms for the tools to be used.		Software Requirements Engineering
4.3.19.2	**Programming Languages:** The selection of suitable programming languages shall be justified for the required Assurance Level$_{SWAL2}$.		Software Safety Requirements Analysis
4.3.19.3	**Compilers considerations:** Compilers mode of use (optimisations, limitations,…) shall be defined$_{SWAL4}$.		Software Requirements Engineering
4.3.19.4	**SW development tool validation:** The context for such a validation shall be defined$_{SWAL1}$. (Validation/certification of compilers/linkers/code generation tools)		Software Requirements Engineering

Objective:	4.3.20	Complexity Constraints	SWAL 2
Activities [EUROCAE09]:			**ISaPro® Mapping:**
4.3.20.1	A level of complexity (as well as selected criteria defining this complexity) shall be defined and measured.		Project Planning

Summary

Table 11 shows the compacted mapping results for the objectives of the primary lifecycle process development to the processes of the ISaPro® framework. The table contains only those ISaPro® processes, which have at least a single mapping to one of the objectives.

ISaPro® Processes [TSS12]	ED-153 Objective N° [EUROCAE09]	Comments
Project Management Lifecycle		
Project Initialisation	4.3.3	4.3.3: Added activity for defining stage gates in project management lifecycle.
Project Planning	4.3.3, 4.3.9, 4.3.10, 4.3.11, 4.3.16, 4.3.17, 4.3.20	4.3.10: Focussing on software standards/rules. Therefore an activity was added for identification of necessary standards and/or rules. 4.3.11: Added activity for defining requirement engineering methods, procedures and the selection of a tool. 4.3.16: Added activity for describing software development lifecycle, including transition criteria for all phases. 4.3.17: Added activity for defining design methods, procedures and the selection of a tool. 4.3.20: Added activity for defining and measuring the complexity of the project.
Project Controlling	4.3.16	
Safety Lifecycle		
Software Safety Requirements Analysis	4.3.19	
Engineering Lifecycle		
Requirements Engineering	4.3.1	
System Design	4.3.2	
Software Requirements Engineering	4.3.3, 4.3.4, 4.3.12, 4.3.13, 4.3.15, 4.3.19	4.3.4: Added activity for defining configuration/adaption data range. 4.3.13: Expanded the resource requirement activity to define a necessary margin for safety purposes.
Software Design	4.3.5, 4.3.14, 4.3.15, 4.3.18	4.3.5: Added activity for definition of software integration. 4.3.14: Added activity especially for real-time design.
Software Component Design	4.3.6, 4.3.15	4.3.6: Expanded activity by mentioning implementation of unit tests.
Software Construction	4.3.15	
Software Integration	4.3.7	
Software Test	4.3.13	
System Integration	4.3.8, 4.3.13	4.3.8: Added activity for documentation of installation procedure and deposition of resources.

Support Processes		
Configuration Management	4.3.3	

Table 11: Mapping of Development Process Objectives

Supporting Lifecycle Processes

Configuration Management

Objective:	5.2.1	Configuration Management Process Implementation	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.1.1	A configuration management plan shall be developed.		Configuration Management
5.2.1.2	The plan shall include, as a minimum: • the configuration management activities; • procedures and schedule for performing these activities; • the organisation(s) responsible for performing these activities; and their relationship with other organisations, such as software development or maintenance; • Software lifecycle environment control management (tools used to develop or verify SW) • Definition of SW lifecycle data (any output relevant to the safety assurance of the software) control management.		Configuration Management
5.2.1.3	The plan shall be documented, placed under configuration management and implemented.		Configuration Management

Objective:	5.2.2	Configuration Identification	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.2.1	A scheme shall be established for identification of software and their versions to be controlled throughout the complete lifecycle of the software.		Configuration Management
5.2.2.2	For each version of all software, the following shall be identified, as a minimum: • the documentation that establishes the baseline; • the version references; • the problem reports list (those already fixed, those fixed in that particular version and those still open if any); • and other identification details.		Configuration Management
5.2.2.3	The items to be configuration-identified shall be identified, along with their associated configuration management level.		Configuration Management

Objective:	**5.2.3**	**Configuration Control**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.3.1	The following shall be performed: identification and recording of change requests; analysis and evaluation of the changes; approval or rejection of the request; and implementation, verification, and release of the modified software.		Change Management
5.2.3.2	An audit trail shall exist, whereby each modification, the reason for the modification, and authorisation of the modification can be traced.		Change Management
5.2.3.3	Control and audit of all accesses to the controlled software that handle safety related functions shall be performed.		Change Management

Objective:	**5.2.4**	**Configuration Status Accounting**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.4.1	Management records and status reports that show the status and history of controlled software including baseline shall be prepared.		Configuration Management
5.2.4.2	Status reports shall include the number of changes for a project, latest software versions, release identifiers, the number of releases, and comparisons of releases.		Configuration Management

Objective:	**5.2.5**	**Configuration Evaluation**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.5.1	The following shall be determined and ensured: the functional completeness of the software against their requirements and the physical completeness of the software (whether their design and code reflect an up-to-date technical description).		Change Management

Objective:	**5.2.6**	**Retrieval & Release Process**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.6.1	A retrieval and release process shall exist.		Configuration Management
5.2.6.2	A retrieval and release process shall be documented.		Configuration Management
5.2.6.3	The release and delivery of software products and documentation shall be formally controlled.		Configuration Management
5.2.6.4	Master copies of code and documentation shall be maintained for the life of the software product.		Configuration Management

Objective:	**5.2.7**	**Use of a CM Tool**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.7.1	A tool shall be used to perform Software configuration management.		Configuration Management

Objective:	**5.2.8**	**Use of a CM Tool (Acquirer Agreement)**	**SWAL 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.8.1	The acquirer shall approve the selected software configuration management tool.		*Not applicable*

Objective:	**5.2.9**	**At Level of SW Component**	**SWAL 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.9.1	The software configuration management shall be performed at the Software Unit level.		Configuration Management

Objective:	**5.2.10**	**Configuration Management Traceability**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.10.1	Software lifecycle data (any output) shall be traceable between versions.		Configuration Management
5.2.10.2	All lifecycle data shall be traceable to the version of software being deployed.		Configuration Management

Objective:	**5.2.11**	**At Level of SW Source Code**	**SWAL 2**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.2.11.1	The software configuration management shall be performed at the Software source code level.		Configuration Management

Quality Assurance Process

Objective:	5.3.1	Process Implementation	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.3.1.1	A quality assurance process tailored to the project shall be established.		Quality Assurance
5.3.1.2	The objectives of the quality assurance process shall be to assure that the software products and the processes employed for providing those software products comply with their established requirements and adhere to their established plans.		Quality Assurance
5.3.1.3	A plan for conducting the quality assurance process activities and tasks shall be defined, implemented, and maintained (including configuration management of evidence records) throughout the relevant parts of the software lifecycle.		Quality Assurance

Objective:	5.3.2	Product Assurance	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.3.2.1	It shall be assured that all the plans required by ED-153 are defined, are mutually consistent, and are being executed as required.		Quality Assurance
5.3.2.2	A Software Conformity review shall be performed.		Quality Assurance

Objective:	5.3.3	Process Assurance	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.3.3.1	It shall be assured that those software lifecycle processes (supply, development, operation, maintenance, and supporting processes including quality assurance) employed for the project adhere to the plans.		Quality Assurance
5.3.3.2	It shall be assured that the internal software engineering practices, development environment and test environment adhere to the plans.		Quality Assurance

Verification Process

Objective:	5.4.1	Verification Process Implementation	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.1.1	A verification process tailored to the software shall be established.		Verification
5.4.1.2	The output of the verification process shall be documented and distributed to the interested parties.		Verification

Objective:	**5.4.2**	**Verification Plan**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.2.1	A verification plan shall be defined.		Verification
5.4.2.2	The plan shall address the lifecycle verification activities and phase outputs subject to verification and related resources, responsibilities, pass fail criteria, methods and schedule.		Verification
5.4.2.3	The plan shall address procedures for forwarding verification reports to the interested parties stating the action to be taken by each party.		Verification
Note: Ensure that the description of the various testing activities and the phase of the SW lifecycle (FAT, SAT, software testing) is included somewhere eg as part of the verification plan. *Note: Objectives regarding the verification of the configuration/adaptation data may be expanded in operation process (see 4.4). The strategy for verifying the appropriate combination of configuration/adaptation data is described in the verification plan.*			

Objective:	**5.4.3**	**Verification of Software Requirements**	**SWAL 3 – 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.3.1	It shall be verified that software requirements are correct and complete$_{SWAL4}$;		Software Requirements Engineering
5.4.3.2	The software requirements shall be verified considering the functional behaviour of the implemented Software complies with the Software Requirements$_{SWAL4}$;		Software Requirements Engineering
5.4.3.3	The software requirements shall be verified considering the timing performances of the implemented software complies with the Software Requirements$_{SWAL4}$;		Software Requirements Engineering
5.4.3.4	The software requirements shall be verified considering the software requirements are consistent, feasible, and verifiable$_{SWAL4}$;		Software Requirements Engineering
5.4.3.5	The software requirements shall be verified considering implemented software robustness to abnormal operating conditions complies with the Software Requirements$_{SWAL3}$;		Software Requirements Engineering
5.4.3.6	The software requirements shall be verified considering external consistency (boundaries) with the system requirements$_{SWAL4}$;		Software Requirements Engineering
5.4.3.7	The software requirements shall be verified considering internal consistency between software requirements$_{SWAL4}$;		Software Requirements Engineering
5.4.3.8	The software requirements shall be verified considering compatibility between implemented software and the HW/SW features of the target computer (system response time, Input/output HW, operation on the target computer)$_{SWAL4}$;		Software Requirements Engineering
5.4.3.9	The software requirements shall be verified considering Software requirements conform to Software requirements standards/rules$_{SWAL4}$;		Software Requirements Engineering
5.4.3.10	The software requirements shall be verified considering algorithms are accurate and correct$_{SWAL3}$;		Software Requirements Engineering
5.4.3.11	The software requirements shall be verified considering the capacity of the implemented software complies with the Software Requirements$_{SWAL3}$;		Software Requirements Engineering
5.4.3.12	The software requirements shall be verified considering the overload tolerance of the implemented Software complies with the Software Requirements$_{SWAL3}$.		Software Requirements Engineering

Objective:	**5.4.4**	**Integration Verification**	**SWAL 2 – 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.4.1	The integration verification shall verify whether the software components have been completely and correctly integrated into the software$_{SWAL3}$.		Software Integration
5.4.4.2	The integration verification shall verify whether the software units have been completely and correctly integrated into the software component$_{SWAL2}$.		Software Integration
5.4.4.3	The integration verification shall verify whether the hardware items, software, and manual operations of the system have been completely and correctly integrated into the system$_{SWAL3}$.		Software Integration
5.4.4.4	The integration verification shall verify whether the integration tasks have been performed in accordance with an integration plan$_{SWAL3}$.		Software Integration
Examples of verification criteria are (especially as far as isolation between software is concerned): • Linking and loading data and memory map • Data control and coupling • Incorrect HW addresses • Memory overlaps • Missing SW components. *Note: Global verification should be performed either through tests or other methods like reviews*			

Objective:	**5.4.5**	**Verification of Software Architectural Design**	**SWAL 3**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
When evaluating the tests, test results to verify the software architectural design, and user documentation:			
5.4.5.1	External consistency with the software requirements (hardware-software compatibility) shall be considered;		Software Design
5.4.5.2	Internal consistency (data flow and control flow) shall be considered;		Software Design
5.4.5.3	Verification coverage of the software architectural design shall be considered;		Software Design
5.4.5.4	Design conformity to Design standards/rules shall be considered;		Software Design
5.4.5.5	Appropriateness of test standards/rules and methods used shall be considered;		Software Design
5.4.5.6	Conformance to expected results shall be considered;		Software Design
5.4.5.7	Feasibility of software design testing shall be considered;		Software Design
5.4.5.8	Feasibility of maintenance shall be considered;		Software Design
5.4.5.9	Verification criteria on which verification completion will be judged shall be considered.		Software Design
5.4.5.10	The results of the evaluations shall be documented.		Software Design
Note: The compliance should be verified according to the the definition of the transition criteria between lifecycle phases (cf SWAL allocation for Development process)			

Objective:	**5.4.6**	**Verification of Detailed Design**	**SWAL 2**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
When evaluating the software code and verification results:			
5.4.6.1	External consistency with the requirements and design of the software (hardware-software compatibility) shall be consider;		Software Component Design
5.4.6.2	Internal consistency between detailed design requirements shall be consider;		Software Component Design
5.4.6.3	Verification coverage of detailed design (units)shall be considered;		Software Component Design
5.4.6.4	Code conforms to Code standards/rules shall be consider;		Software Component Design
5.4.6.5	Verification of the coverage of the software structure (statement coverage) shall be consider - see note below this table;		Software Component Design
5.4.6.6	Appropriateness of coding methods and standards/rules used shall be consider;		Software Component Design
5.4.6.7	Feasibility of software code verification shall be consider;		Software Component Design
5.4.6.8	Feasibility of maintenance shall be consider.		Software Component Design
5.4.6.9	The results of the evaluations shall be documented.		Software Component Design
Note: Global verification should be performed either through tests or other methods like reviews or other means…..			

Objective:	**5.4.7**	***Removed***	**-**

Objective:	**5.4.8**	**Verification of Executable Code**	**SWAL 1**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.8.1	Executable code and verification results shall be evaluated considering the criteria listed below: • External consistency with the code of the software (eg is the compiler generating an appropriate executable or object code?); • Internal consistency between exe requirements (eg: is the compiler always generating the same executable or object code for the same source?); • Verification of the translation of the software source code into object code (eg is the compiler generating additional and unnecessary executable or object code, such as dead executable code?); • Feasibility of executable verification; • Verification of software structure (MC/DC).		Software Construction
5.4.8.2	The results of the evaluations shall be documented.		Software Construction

Objective:	**5.4.9**	**Data Verification**	**SWAL 2**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.9.1	The data structures specified during detailed design shall be verified for: • Completeness • Self-Consistency • Protection against alteration or corruption		Software Component Design

Objective:	**5.4.10**	**Traceability**	**SWAL 1 – 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.10.1	Traceability shall be verified between System and Software requirements$_{SWAL4}$		Software Requirements Engineering
5.4.10.2	Traceability shall be verified between Software requirements and Software Architectural Design$_{SWAL3}$		Software Design
5.4.10.3	Traceability shall be verified between Software Architectural Design and Detailed Design$_{SWAL2}$		Software Component Design
5.4.10.4	Traceability shall be verified between Software Detailed Design and Executable Code$_{SWAL1}$		Software Construction
5.4.10.5	Traceability shall be verified between verification evidence and Software Requirements$_{SWAL4}$		Software Test
5.4.10.6	Traceability shall be verified between safety assurance evidence and the version of the software being deployed$_{SWAL4}$		Software Integration

Objective:	**5.4.11**	**Complexity Measures**	**SWAL 2**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.11.1	It shall be demonstrated that the measured complexity is within the defined threshold by: • analysing the measures, and • applying corrective actions.		System Integration
5.4.11.2	If value exceeds thresholds (to be defined), a justification shall be provided.		System Integration

Objective:	5.4.12	Verification of Verification Process Results	SWAL 2 – 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.12.1	Verification cases, procedures and results shall be verified, so that: • Verification procedures are correct and complete and discrepancies are justified$_{SWAL4}$ • Verification results are correct and complete and discrepancies are justified$_{SWAL4}$ • Verification of the software requirements verification cases, procedures and results is correct and complete and discrepancies are justified$_{SWAL4}$ • Verification of the software design (architectural level) verification cases, procedures and results is correct and complete and discrepancies are justified$_{SWAL3}$ • Verification of the software design (detailed design) verification cases, procedures and results is correct and complete and discrepancies are justified$_{SWAL2}$ • Verification of the software integration verification cases, procedures and results is correct and complete and discrepancies are justified$_{SWAL3}$ • Verification of the software data verification cases, procedures and results is correct and complete$_{SWAL3}$ • Verification of the traceability verification procedures and results is correct and complete and discrepancies are justified$_{SWAL4}$		Verification
NOTE: Verification may be performed through inspection, analysis, demonstration or a combination of them all throughout the lifecycle.			

Objective:	5.4.13	Verification of Retrieval and Release Process	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.4.13.1	The Software Retrieval and release process shall be verified.		Configuration Management

Validation Process

According EUROCAE [EUROCAE09] the validation process is considered as out of scope for the guidance ED-153.

Joint Review Process

Objective:	5.6.1	Process Implementation	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.6.1.1	Periodic reviews shall be held at predetermined milestones as specified in the project plan(s).		Project-Controlling
5.6.1.2	The review results shall be documented and distributed.		Project-Controlling

Objective:	5.6.2	Project Management Reviews	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.6.2.1	Project status shall be evaluated relative to the applicable project plans, schedules, standards/rules, transition criteria and guidelines.		Project-Controlling

Objective:	5.6.3	Technical Reviews	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
5.6.3.1	Technical reviews shall be held to evaluate the software products or services under consideration.		Configuration Management

Summary

Table 12 depicts the compacted mapping results of the objectives for the supporting processes. The table contains only those ISaPro® processes, which have at least a single mapping to one of the safety assurance system objectives.

ISaPro® Processes [TSS12]	ED-153 Objective N° [EUROCAE09]	Comments
Project Management Lifecycle		
Project Controlling	5.6.1, 5.6.2	
Engineering Lifecycle		
Software Requirements Engineering	5.4.3, 5.4.10	5.4.3: Added activities for verification of considered requirements regarding robustness, used algorithms, capacity and overload tolerance of the implemented software.
Software Design	5.4.5, 5.4.10	5.4.5: Added activities for verification of coverage, feasibility study, verification criteria and documentation of the whole software design evaluation.
Software Component Design	5.4.6, 5.4.9, 5.4.10	5.4.6: Added activities for verification of coverage, feasibility study and documentation of the whole software component design evaluation.
Software Construction	5.4.8, 5.4.10	5.4.8: Added activity for evaluation of executable code and verification results.
Software Integration	5.4.4, 5.4.10	5.4.4: Expansion of the activity including completeness and correctness. 5.4.10: Added activity for ensuring traceability between safety assurance and the version of software.
Software Test	5.4.10	
System Integration	5.4.11	5.4.11: Added activity for verifying the previously defined complexity of the project.

Support Processes		
Configuration Management	5.2.1, 5.2.2, 5.2.4, 5.2.6, 5.2.7, 5.2.9, 5.2.10, 5.2.11, 5.4.13, 5.6.3	5.2.1: Expanded activity because of the strong focus on software. 5.2.4: Added activity for creating management records and status reports. 5.2.6: Added activity for a retrieval and release process for configuration items. 5.2.7: Added activity for using a configuration management tool. 5.2.9 / 5.2.11: Dependent on SWAL level, the configuration management process has to be performed on different levels. 5.2.10: Added activity for ensuring traceability between software lifecycle data and deployed versions.
Quality Assurance	5.3.1, 5.3.2, 5.3.3	
Verification	5.4.1, 5.4.2, 5.4.12	5.4.12: Expanded activity by adding the verification of the verification.
Change Management	5.2.3, 5.2.5	5.2.3: Added activity for a change management audit trail, including controlling of access to the software.

Table 12: Mapping of Supporting Processes Objectives

Organisational Lifecycle Processes

Management Process

Objective:	**6.1.1**	**Management Process Implementation Initiation & Scope Definition**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
6.1.1.1	A management process tailored to the project shall be defined.		Project Initialisation
6.1.1.2	The output of the management process shall be documented and distributed.		Project Initialisation
6.1.1.3	The management process shall be initiated by establishing the requirements of the process to be undertaken.		Project Initialisation
6.1.1.4	The manager shall establish the feasibility of the process by checking that the resources (personnel, materials, technology, and environment) required to execute and manage the process are available, funded, adequate, and appropriate and that the timescales to completion are achievable.		Project Initialisation

Objective:	**6.1.2**	**Planning**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
6.1.2.1	The manager shall prepare the plans for execution of the process.		Project Planning
6.1.2.2	The plans associated with the execution of the process shall contain descriptions of the associated activities and tasks and identification of the software products that will be provided.		Project Planning
6.1.2.3	These plans shall include, as a minimum, the following: • Schedules for the timely completion of tasks; • Estimation of effort; • Adequate resources needed to execute the tasks; • Allocation of tasks (including who, what and when); • Assignment of responsibilities; • Quantification of project risks associated with the tasks or the process itself; • Quality control measures to be employed throughout the process; • Costs associated with the process execution; • Provision of environment and infrastructure.		Project Planning

Objective:	**6.1.3**	**Execution & Control**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
6.1.3.1	The manager shall initiate the implementation of the plan to satisfy the objectives and criteria set, exercising control over the process.		Project Controlling
6.1.3.2	The manager shall monitor the execution of the process, providing both internal reporting of the process progress and external reporting to the acquirer as defined in the contract.		Project Controlling
6.1.3.3	The manager shall investigate, analyse, and resolve the problems discovered during the execution of the process.		Project Controlling

Objective:	**6.1.4**	**Review & Evaluation**	**SWAL 4**
Activities [EUROCAE09]:			**ISaPro® Mapping:**
6.1.4.1	The manager shall ensure that the software lifecycle data is evaluated for satisfaction of requirements.		Project Controlling
6.1.4.2	The manager shall assess the evaluation results of the software products, activities, and tasks completed during the execution of the process vis-à-vis the achievement of the objectives and completion of the plans.		Project Controlling

Objective:	6.1.5	Closure	SWAL 4
Activities [EUROCAE09]:			**ISaPro® Mapping:**
6.1.5.1	When all software products, activities, and tasks are completed, the manager shall determine whether the process is complete taking into account the criteria as specified in the contract or as part of organisation's procedure.		Project Close-Down
6.1.5.2	The manager shall check the results and records of the software products, activities, and tasks employed for completeness.		Project Close-Down
6.1.5.3	These results and records shall be archived in a suitable environment as specified in the contract.		Project Close-Down

Summary

Table 13 shows the compacted mapping results of the management process. The table contains only those ISaPro® processes, which have at least a single mapping to one of the safety assurance system objectives.

ISaPro® Processes [TSS12]	ED-153 Objective N° [EUROCAE09]	Comments
Project Management Lifecycle		
Project Initialisation	6.1.1	
Project Planning	6.1.2	6.1.2: Added activities for describing project deliverables and the provision of environment and infrastructure.
Project Controlling	6.1.3, 6.1.4	6.1.4: Added activity for evaluation of software lifecycle data and results of activities and tasks.
Project Close-Down	6.1.5	6.1.5: Added activities for determination of the achieved project results and records, when the project is finished.

Table 13: Mapping of Management Process Objectives

Annex B: Adapted Integrated Process Model

This appendix includes all EUROCAE ED-153 (see chapter 3) required activities per each process of the ISaPro® framework (see chapter 4). In addition the appendix comprises the linking of the activities to the phases of the agile procedure model (see chapter 8) where these activities have to be conducted.

The activities are composed of the following:

- Activities already intended in ISaPro® and perfectly fitting to one or more objectives
- Activities already intended in ISaPro®, which are extended or amended in order to fit to one or more objectives
- Own activities, which were identified during analysis of the objectives and therefore added to this adapted process model

Already intended activities are written in *italic* in order to indicate that these are developed by Tschürtz et al. [TSS12]. Further amendments and identified activities during analysis of ED-153 [EUROCAE09] are written in ordinary style. A SWAL (see chapter 3.2) added as subscripted information to an activity indicates that this activity has to be performed only when the mentioned or a more rigorous SWAL is required. Numbers in brackets refer to the objective N° of ED-153 (see Annex A). Activities, where no objective number is available, are not specifically mentioned by the guidance, but are valuable in order to assure safety compliance.

The linking to the agile procedure is done by marking each activity with a specific symbol, where each symbol represents one phase of the procedure model:

- ○ Pre-game phase (see chapter 8.2)
- □ Iteration-driven phase (see chapter 8.3)
- △ Spin-off phase (see chapter 8.4)
- ◇ Wrap-up phase (see chapter 8.5)

Project Management Lifecycle

Project Initialisation:

- *Enquiry of business requirements* (3.1.1.1, 3.1.1.2, 6.1.1.3) ○
- *First estimation of safety integrity level* ○
- *Definition of project strategy* (6.1.1.1) ○
- *Preparation of rough schedule, including milestones* (4.3.3.2) ○
- Definition of stage gates and their corresponding criteria to pass (4.3.3.2) ○
- *Definition of the number of necessary audits, assessments and reviews* (4.3.3.2) ○
- *Examination of feasibility (technical and organisational)* (6.1.1.4) ○
- *Creation of first draft of project management plan* (6.1.1.2) ○
- *First estimation of project risks* ○

Project Planning:

- *Identification of required norms, standards and rules* (3.1.4.1, 4.3.9.2) ○
  - Safety standard specifications, software requirement, design$_{SWAL3}$ and coding $_{SWAL2}$ rules, etc. (4.3.10.1, 4.3.10.2, 4.3.10.3)
- Definition and measuring of project complexity$_{SWAL2}$ (4.3.20.1) ○
- *Definition of safety integrity level* ○
- Creation and *integration of safety planning* (3.2.1.1, 3.2.2.1) ○
- *Definition of way of working and derivation of project life cycles* (3.2.1.1, 4.3.3.1, 4.3.3.3) ○
  - Definition of software development lifecycle$_{SWAL3}$ and transition criteria for development, requirements and verification phases$_{SWAL3}$ / all phases$_{SWAL2}$ (4.3.16.1, 4.3.16.3, 4.3.16.4)
  - Definition of requirements, development methods, procedures and identification and selection of a requirement specification tool (4.3.11.1)
  - Definition of design methods, procedures, and identification and selection of a design tool$_{SWAL3}$ (4.3.17.1)
- *Planning of productivity* (4.3.9.1, 4.3.9.2) ○
- *Estimation of work packages* (6.1.2.2, 6.1.2.3) ○ □ △ ◇
- Definition of deliverables (6.1.2.2) ○
- *Planning of time schedule* (6.1.2.3) ○ □ △ ◇
- *Definition of project organisation* (4.3.9.2, 6.1.2.3) ○
- *Identification of project risks and initialisation of risk management* (6.1.2.3) ○ □
- *Planning of project controlling meetings* (6.1.2.3) ○ □
- Provision of environment and infrastructure (6.1.2.3) ○ □
- *Finishing of project management plan* (4.3.9.3, 6.1.2.1) △ ◇
- *Reviews of all created documents* (especially project management plan) (3.2.3.1) ○
- *Gain commitment from stakeholders and release of project management plan* (3.2.4.1) ○ □

Project Controlling (5.6.1.1, 6.1.3.1):

- *Identification of status quo* (3.4.3.1, 5.6.2.1, 6.1.3.2) □ △ ◇
- Comparison of status quo to planned one and identification of deviations (particular focus on software safety assessment activities) (3.4.3.1, 5.6.2.1, 6.1.3.2) □ △ ◇
- Evaluation of software lifecycle data and assessment of completed software products, activities and tasks (6.1.4.1, 6.1.4.2) □ △ ◇
- *Monitoring of involvement of stakeholders* (6.1.3.2) □ △ ◇
- *Monitoring of project risks* □ △ ◇
- *Assessment and evaluation of milestones*$_{SWAL3}$ (4.3.16.2) □ △ ◇
- *In the case of a deviation: execution of cause and consequences analysis* (6.1.3.3) □ △ ◇

- *Planning of steering measures* (6.1.3.3) ☐ △ ◇
- *Creation of report* (5.6.1.2, 6.1.3.2) ☐ △ ◇
- *Updating the project plans* ○ ☐ △ ◇

Project Close-Down:

- Ensuring of completeness for all software products, activities, tasks and their results and records (6.1.5.1, 6.1.5.2) △ ◇
- Archiving of all results and records in a suitable environment (6.1.5.3) △ ◇

Safety Lifecycle

Preliminary Hazard Identification (PHI):

- *Definition of safety goals* ○
- *Creation of a preliminary hazard list based on information of business requirements and rough concept* ○

Functional Hazard Evaluation (FHE):

- *Identification of hazards on the basis of the technical concept* (3.3.1.2, 3.3.2.2) ○ ☐
- *Analysis of failure modes relating to the operation of the system (using of error and failure checklists)* (3.3.1.1) ○ ☐
- *Expansion of these checklists* (3.3.1.2) ○ ☐
- *Analysis of hypothetical failure mode*s (3.3.1.1) ○ ☐
- *Analysis of effects and consequences of those failure modes* (3.3.2.1) ○ ☐
  - *Dangerous system states, accidents, disadvantages*
- *Definition and review of safety requirements on system level* ○ ☐

Preliminary System Safety Evaluation (PSSE):

- *Ensuring that system safety requirements are realised by the system design* ○ ☐
- *Identification of newly raised hazards by the design itself* ○ ☐
- *Expansion of the hazard list created in previous processes* ○ ☐

Software Safety Requirements Analysis (SSRA):

- *Analysis of software hazards* (including evaluation) (3.3.2.1, 3.3.2.2) ○ ☐
- *Analysis of causes* and creation of a list consisting of common modes of failure (3.3.1.1, 3.3.1.2) ○ ☐
- Update of initial risk assessment and mitigation process based on executed analyses (3.3.3.1) ○ ☐
- *Identification of safety-critical interfaces* ○ ☐
- *Identification of hazard mitigation requirements* (3.4.2.1) ○ ☐

- *Identification of safety-critical software requirements* based on functional and non-functional software requirements (3.3.4.1) ○ □
- *Specification of software safety integrity level (SIL)* (3.3.4.1) ○
- *Definition of necessary tools for software development (programming language, compiler, etc.)*$_{SWAL2}$ (4.3.19.2) ○
- *Creation and release of software safety requirements* (3.3.4.1) ○ □

Software Safety Design Analysis (SSDA):

- Ensuring of consistency to risk assessment and mitigation identification during preliminary system safety evaluation (PSSE) (3.1.5.1) ○ □
- *Analysis of timing, throughput and sizing of software* ○ □

System Safety Evaluation (SSE):

- *Definition of required methods related to implementation* ○
- *Ensuring of accomplishment of required tests* □ △ ◇

Engineering Lifecycle

Concept:

- *Definition of yet is to be developed system* (3.1.1.3) ○
- *Definition of system boundaries* (3.1.1.4) ○
- *Definition of required safety standard specifications* (3.1.3.1) ○
- *Identification of system environment and analysis of system behaviour* (3.1.2.1) ○
- *Definition of system interfaces* (3.1.1.5) ○

System Requirements Engineering:

- *Definition of interface requirements* (4.3.1.1) ○ □
- *Definition of system requirements* (4.3.1.1) ○ □
  - *Function requirements*
  - *Non-functional requirements*
  - *Safety requirements (derived from FHE)*
- *Analysis of system requirements* (4.3.1.1) ○ □
  - *Prioritisation of requirements*
  - *Examination as to whether requirements are correct, complete, consistent, feasible and testable*
  - *Identification of operation environment*
- *Evaluation and actualisation of system requirements* (4.3.1.1) ○ □
  - *Defined process for handling of changes*
  - *Evaluation of human interface*
- *Ensuring consistency and traceability* ○ □

System Design:

- *Definition of system architecture* (4.3.2.1) ○ ☐
- *Mapping of defined system requirements to system elements* (4.3.2.1) ○ ☐
  - *Mapping of functional, non-functional and safety requirements*
- *Definition of necessary interfaces* (4.3.2.1) ○ ☐
  - *Designing and documentation of internal and external interfaces*
- *Analysis of preliminary system safety assessment (PSSE) and incorporation of inevitable design changes* (4.3.2.1) ○ ☐
- *Ensuring consistency and traceability* ○ ☐

Software Requirements Engineering:

- *Execution of requirements analysis* (4.3.4.1) ○ ☐
  - *Collecting, analysing, categorising, prioritising and documenting requirements*
  - *Analysis as to whether requirements are correct, complete, consistent, feasible and testable* (3.4.1.1, 5.4.3.1, 5.4.3.4, 5.4.3.6, 5.4.3.7)
  - Use of a requirement specification tool (4.3.12.1)
- *Specification of software requirements* (4.3.4.2) ○ ☐
  - *Functional* (5.4.3.2) *and non-functional software requirements*
- *Specification of performance criteria* (4.3.4.2, 5.4.3.3) ○ ☐
  - *Speed, response time, recovery time, etc.*
- Specification of necessary margin of resources (e.g. memory, CPU load, etc.) for safety purposes (4.3.13.1) ○ ☐
- Specification of robustness, capacity and overload tolerance of the software$_{SWAL3}$ (5.4.3.5, 5.4.3.11, 5.4.3.12) ○ ☐
- Specification of correct and accurate algorithms used in the software$_{SWAL3}$ (5.4.3.10) ☐
- *Definition of operational environment and derived requirements* (5.4.3.8) ○ ☐
- Definition of configuration/adaption data ranges (4.3.4.2) ☐
- *Definition of general requirements (standard specifications, programming languages (and their justification to the SWAL$_{SWAL2}$), coding guidelines tools, etc.)* (4.3.3.3, 4.3.4.3, 4.3.19.1, 4.3.19.2, 5.4.3.9) ○
  - Verification of software development tools$_{SWAL1}$ (4.3.19.4)
- *Analysis of software safety requirements* ○ ☐
- *Ensuring of consistency to system requirements* (4.3.4.2, 4.3.15.1, 5.4.10.1) ○ ☐ △ ◇
- *Execution of inspections and reviews* (3.4.1.1, 4.3.4.2) ☐

Software Design:

- *Characterisation of interfaces, dependencies between components*$_{SWAL3}$ (4.3.14.1) ○ ☐

- *Definition of software architecture*$_{SWAL3}$ (4.3.5.1) ○ ☐
  - *Use of standard architectures or analysing and evaluation of different architectures*
  - *Documentation of the basis of decision-making*
  - Use of design tool (4.3.17.1)
  - Execution of feasibility study on software design testing and maintenance (5.4.5.7, 5.4.5.8)
- Definition of software integration$_{SWAL3}$ (4.3.5.1) ○ ☐
- Definition of real-time features (if applicable)$_{SWAL3}$ (4.3.14.1) ○ ☐
  - Tasks and runtime aspects, interruptions, treatment and propagation of errors, data management and initialisation/stop of software
- *Definition interfaces*$_{SWAL3}$ (4.3.5.1) ○ ☐
  - *Specification of internal and external interfaces*
  - *Definition of input and output data*
- *Analysis of testability*$_{SWAL3}$ (5.4.5.3, 5.4.5.5) ○ ☐ △ ◇
  - *Check design for correctness (design reviews)*$_{SWAL3}$ (5.4.5.4, 5.4.5.7, 5.4.5.8) ☐
  - Definition of verification coverage and criteria of completeness$_{SWAL3}$ (5.4.5.3, 5.4.5.9) ☐ △ ◇
  - Implementation of unit tests$_{SWAL2}$ (4.3.6.1) ☐
  - *Development of test concept and creation of test plan (activities of test group)*$_{SWAL3}$ (5.4.5.5, 5.4.5.9) ○ ☐ △ ◇
- *Ensuring consistency to system requirements and system design (traceability)*$_{SWAL3}$ (4.3.15.2, 5.4.5.1, 5.4.5.2, 5.4.10.2) ☐
- *Ensuring conformance to safety standard specifications and expected results*$_{SWAL3}$ (5.4.5.6) ☐
- Documentation of the results of software design phase$_{SWAL3}$ (5.4.5.10) ○ ☐

Software Component Design:

- *Development of component design*$_{SWAL2}$ (4.3.6.1) ○ ☐
  - *Grouping of software components into programming elements (modules)* ☐
  - *Describing of module functionalities, their dependencies and interfaces* (5.4.6.2) ☐
  - *Definition of detailed functions, used algorithms, in- and output data, used data formats, memory usage, etc.* (5.4.6.6) ☐
  - *Definition of required standard specifications and general requirements* (5.4.6.4) ○ ☐
- Execution of feasibility study on software code verification and maintenance$_{SWAL2}$ (5.4.6.7, 5.4.6.8) ☐
- Definition of verification coverage$_{SWAL2}$ (5.4.6.3) ☐
- *Ensuring consistency to software architecture (traceability)*$_{SWAL2}$ (5.4.6.1, 5.4.10.3) ☐

- Documentation of the results of software detailed design phase$_{SWAL2}$ (5.4.6.9) ☐
- *Execution of reviews*$_{SWAL2}$ (5.4.6.5) ☐

Software Construction:

- Evaluation of executable code, including internal and external consistency, the verification of the translation from source into object code, the feasibility of the executable and the verification of the software structure$_{SWAL1}$ (5.4.8.1) ☐
- *Compliance with documentation guidelines*$_{SWAL1}$ (5.4.8.2) ☐
- *Put source code under configuration management*$_{SWAL4}$ ☐
- *Ensuring traceability between software component design and code*$_{SWAL2}$ *and between code and executable*$_{SWAL1}$ (4.3.15.3, 4.3.15.4, 5.4.10.4) ☐

Software Component Test:

- *Development of component test specifications* ☐
- *Execution of component tests* ☐ △ ◇
  - *Documentation of test results and failures*
- *Execution of regression tests* ☐ △ ◇

Software Integration:

- *Development and planning of integration strategy*$_{SWAL3}$ (4.3.7.1, 4.3.7.2) ○ ☐
  - *Planning of integration activities including their corresponding software components*
- *Development of integration tests and their specifications*$_{SWAL3}$ (4.3.7.2) ☐
- *Execution of documentation reviews and inspections*$_{SWAL3}$ (4.3.7.3) ☐
- *Integration of software modules according to an integration plan*$_{SWAL3}$ (5.4.4.4) ☐ △ ◇
- *Testing of integrated software modules* ☐ △ ◇
  - Ensuring completeness and correctness of integration of software units into software components$_{SWAL3}$ and further into integrated software$_{SWAL2}$ (5.4.4.1, 5.4.4.2) ☐ △ ◇
  - Ensuring completeness and correctness of integration hardware and software$_{SWAL3}$ (5.4.4.3) ☐ △ ◇
- *Ensuring consistency between software design and software integration* ☐ △ ◇
- Ensuring traceability between safety assurance and the version of software being deployed (5.4.10.6) ☐ △ ◇

Software Test:

- *Development of test strategy and plan* ○ ☐ △ ◇
- *Creation of test specification based on requirements specification* (5.4.10.5) ☐
  - *Definition of test procedures, test cases and test data*

- *Execution of software tests* (3.4.4.1) ☐ △ ◇
  - *Testing software against verification criteria*$_{SWAL3}$ (4.3.13.2)
  - *Documentation of test results and failures*
- *Execution of regression tests* ☐ △ ◇
- *Execution of document reviews* ☐ △ ◇

System Integration:

- *Development and planning of integration strategy* (4.3.8.1) ○ ☐ △ ◇
- Deposition of software resource file and documentation of installation procedure (4.3.8.2) ☐
- *Integration of system elements* ☐ △ ◇
- *Development of tests for system elements*$_{SWAL3}$ (4.3.13.2) ☐
- Ensuring that previously defined complexity is met$_{SWAL2}$ (5.4.11.1, 5.4.11.2) ☐ △ ◇
- *Ensuring consistency (traceability)*$_{SWAL2}$ (5.4.11.1, 5.4.11.2) ☐ △ ◇

System Test:

- *Development of test strategy and plan* ○ ☐ △ ◇
- *Execution of system tests* ☐ △ ◇
  - *Documentation of test results and failures*
- *Execution of regression tests* ☐ △ ◇
- *Execution of document reviews* ☐ △ ◇

Supporting Processes

Quality Assurance:

- *Development of approach* (5.3.1.1) ○
  - *Definition of target items (work products, process step, etc.)* (5.3.1.2)
  - *Definition according to which goals are tested* (5.3.1.2)
  - *Planning of quality assurance activities* (5.3.1.3)
- Ensuring that software safety assessment results are documented and disseminated to impacted parties (3.5.1.1, 3.5.3.1) ☐ △ ◇
- *Definition of quality recordings* (5.3.1.2) ○
- *Execution of compliance tests (quality audits)* (5.3.2.1, 5.3.2.2, 5.3.3.1, 5.3.3.2) ☐ △ ◇
  - *Documentation and dissemination of results*
  - *Evaluation of deviations and triggering of corrective measures*

Verification:

- *Development of verification strategy* (5.4.1.1) ○ □ △ ◇
  - *Definition of used methods, techniques and tools* (5.4.2.2)
  - *Definition of work products and processes, which should be verified* (5.4.2.2)
  - *Planning of verification activities* (5.4.2.1)
- *Development of verification criteria* (5.4.2.2) □ △ ◇
- Ensuring verification of verification (5.4.12.1) □ △ ◇
- *Execution of verification and documentation of results* (5.4.1.2) □ △ ◇
  - *Dissemination of results to relevant stakeholders* (5.4.1.2, 5.4.2.3)

Configuration Management:

- *Development of configuration management strategy and plan* (5.2.1.1, 5.2.1.2) ○
- Definition of software lifecycle environment control management and software lifecycle data (5.2.1.2) ○ □
- *Identification of configuration elements and their required degree of maturity* (especially software safety documents, technical reviews and inspections and the configuration management plan itself) (3.5.2.1, 4.3.3.1, 5.2.1.3, 5.2.2.1, 5.2.2.3) ○ □
- Definition of a retrieval and release process, which is documented and formally controlled and which has master copies that are maintained for the whole product life cycle (5.2.6.1, 5.2.6.2, 5.2.6.3, 5.2.6.4) ○ □ △ ◇
- Use of a configuration management tool (5.2.7.1) ○
- Definition of the depth level, where configuration management is performed (either on software unit_{SWAL3} or software source code_{SWAL2} level) (5.2.9.1, 5.2.11.1) ○
- *Execution of reviews* (5.6.3.1) □ △ ◇
- *Creation of baselines* (5.2.2.2) □ △ ◇
- *Describing of configuration elements* (especially versions, etc.) (5.2.2.2) □ △ ◇
- *Steering of changes* (5.2.2.2) □ △ ◇
- Ensuring traceability of software lifecycle data and versions of software being deployed (5.2.10.1, 5.2.10.2) □ △ ◇
- Providing management records and status reports on controlled software, including baselines and information on release process (5.2.4.1, 5.2.4.2) □ △ ◇

Change Management:

- Set up of a change management audit trail including details on each modification and access to controlled software (5.2.3.2, 5.2.3.3) □ △ ◇
- *Discussion of change requests* (5.2.3.1) □
- *Analysis of change requests by experts* (5.2.3.1, 5.2.5.1) □
- *Rejection or approval of change request through change control board* (5.2.3.1) □

Compliance Analysis Results

Based on the evaluation of compliance (see chapter 8.6) of the agile procedure model to the adapted ISaPro® framework, the following analysis has be performed:

How many activities have to be achieved…

- …per lifecycle of the ISaPro® framework?
- …per agile procedure model phase?
- …per SWAL?

The following analysis results considered only major activities of the adapted ISaPro® framework in Annex B. If one activity is required for multiple SWAL, but in different granularity, this activity is mapped to the less rigorous SWAL.

Project Management Lifecycle					
Phase	**SWAL 4**	**SWAL 3**	**SWAL 2**	**SWAL 1**	**Sum**
Pre-game	24		1		25
Iteration-driven	15	1			16
Spin-off	13	1			14
Wrap-up	13	1			14

Table 14: Analysis Results of Project Management Lifecycle

Safety Lifecycle					
Phase	**SWAL 4**	**SWAL 3**	**SWAL 2**	**SWAL 1**	**Sum**
Pre-game	22		1		23
Iteration-driven	19				19
Spin-off	1				1
Wrap-up	1				1

Table 15: Analysis Results of Safety Lifecycle

Engineering Lifecycle					
Phase	**SWAL 4**	**SWAL 3**	**SWAL 2**	**SWAL 1**	**Sum**
Pre-game	25	9	1		35
Iteration-driven	36	17	9	2	64
Spin-off	15	3	2		20
Wrap-up	15	3	2		20

Table 16: Analysis Results of Engineering Lifecycle

Supporting Processes					
Phase	**SWAL 4**	**SWAL 3**	**SWAL 2**	**SWAL 1**	**Sum**
Pre-game	8	1			9
Iteration-driven	19				19
Spin-off	14				14
Wrap-up	14				14

Table 17: Analysis Results of Supporting Processes