Tampereen teknillinen yliopisto. Ohjelmistotekniikan laitos. Raportti 14
Tampere University of Technology. Department of Software Systems. Report 14

Matti Vuori

Agile Development of Safety-Critical Software

Tampere University of Technology. Department of Software Systems
Tampere 2011

ISBN 978-952-15-2595-7

ISSN 1797-836X

Abstract

Agile software development has gained an enormous success in all kinds of product and system development. It has been expected to provide more control for the development process and to be able to deliver value to customers and developers earlier and to be able to meet challenges in changing requirements more easily than previous process lifecycle models. One area where implementation of agile processes still needs a lot of work before becoming a well understood practice is the development of safety-critical software. This paper analyses the agile principles and processes and gives guidance on how organizations could change their processes to a more agile way without risking the safety or marketability of the products or causing increased product and liability risks. In this report, IEC 61508 standard series was selected as basis for the requirements of safety-critical development.

Acknowledgements

The author would acknowledge partial funding from Tekes and the following companies participating in the Ohjelmaturva project: ABB, Bronto Skylift, EPEC, John Deere Forestry, Konecranes, Metso, Safety Advisor, Sandvik Mining and Construction, and Sundcon.

Contents

1 Introduction

There is a tendency for companies to transform their software and product development practice into more a incremental form, using special agile software development lifecycle and project management models. Incremental development processes have been used previously, but agile processes add to these new project planning, management and execution principles. Still, the main value of agile processes comes from controlled increments and releases, which they produce more often than previous models. Controlled releases should especially help in getting feedback from the customer and managing project risks. (Note: it is assumed that the reader of this report is somewhat familiar with the concepts of agile software development; if that is not the case, the Wikipedia article below about agile development can be a good starting point to familiarise oneself with the topic http://en.wikipedia.org/wiki/Agile_software_development.)

Larman (2004) in his book lists the following key motivations for iterative development:

- Iterative development is lower risk; the waterfall is higher risk.
- Early risk mitigation and discovery.
- Accommodates and provokes early change; consistent with new product development.
- Manageable complexity.
- Confidence and satisfaction from early, repeated success.
- Early partial product.
- Relevant process tracking; better predictability.
- Higher quality; less defects.
- Final product better matches true client desires.
- Early and regular process improvement.
- Communication and engagement required.
- IKIWISI required [IKIWISI = I'll Know It When I See It]

All these are benefits that companies seek when starting to use agile methods. However, the modern agile processes promise even more benefits compared to previous iterative/incremental models, especially the following:

- Shorter time to the first releases and releases more often.
- Reduced amount of process and project documentation, yet better communication and thus a smoother process.
- Increased customer participation.

What the most important benefits are, depends on the type of development; mainly, whether it is customer-oriented development of tailored systems or mass-market oriented development. Sometimes the approaches can be combined if the products are developed for small key clientele and then the offering is targeted on mass markets. But still, the approaches and needs are different.

In the development of tailored systems, the customer's essential needs that agile can bring benefits to are: getting an early release and understanding of the system by using it, getting regular releases at a sensible pace so that the new system can be learned and all necessary adaptations can be made in time, and making changes to plans during the process in a flexible way. In mass-market product development, needs may be more based on the manufacturer's desire to control risks, be fast in responding to competition and emerging market needs.

And the goal of reaching these goals influences how companies approach the development process. Some companies may start the agile process with a very vague idea of a concept, whereas some may see it predominantly as a way to make software production more controlled, and yet others simply aim to get a row of productive solutions, new releases to customers. In fact, release orientation and release readiness are seen as a key element of agile approaches.

Agile development has received critique. Moser et al (2007) write: “Although agile processes and practices are gaining more importance in the software industry there is limited solid empirical evidence of their effectiveness”. Coplien (2011) looks back on the development of agile processes and notes that “However, as with most trademark-able labels, manifestos, and other documented ideals, the reality of the trumpeted practice often missed the mark. “Agile” became a label for a wide collection of otherwise unrelated practices, a collecting point that empowered people to justify their favourite practice.” And Kruchten (2011) documents topics that the agile community is not really willing to tackle for a variety of reasons:

1. Commercial interests censoring failure; 2. Pretending agile is not a business; 3. Failure to dampen negative behaviour; 4. Context and Contextual applicability (of practices; 5. Context gets in the way of dogma; 6. Hypocrisy; 7. Politic; 8. Anarchism; 9. Elitism; 10. Agile alliance; 11. Certification (the “zombie elephant”); 12. Abdicating responsibility for product success (to others, e.g., product owners); 13. Business value; 14. Managers and management are bad; 15. Culture; 16. Role of architecture and design; 17. Self-organising team; 18. Scaling naïveté (e.g., scrum of scrums); 19. Technical debt; 20. Effective ways of discovering info without writing source code.

Note that the list is not based on thorough analysis, but brainstorming at an agile experts' meeting to celebrate 10 years of agile having passed since formulating the Agile Manifesto that defines the agile principles.

This being the situation, the agile approach and agile practices need to be chosen very carefully in a company, and an experienced organisation needs to rely on its own engineering sense to decide on its processes.

For some time, private and public organisations have been replacing their waterfall models or even incremental models with various agile project models and their corresponding practices. This has been going on in all kinds of development domains, simple and complex systems. Agile has mostly been used in small projects; its application in large projects is still a research issue (see, for example, Rohunen et al (2010)). A company named VersionOne has published annual surveys on adaptation of agile development (5th Annual State of Agile Development Survey, 2010), the reports of which contain plenty of detailed information. Still, there are some development contexts where there is still not enough understanding of how agile processes can be utilised so that they bring benefits and do not endanger the quality of operation and products and a product business or customers' operations.

Development of safety-critical systems is one such area. It should be noted that it is not a heterogeneous area, but consists of many different development cultures, defined, for example, by:

- Type of product and system – from medical devices to machines to nuclear power plants.
- The role of software in the system – is it mainly a software-based system or is software still only in a restricted role and the product is perceived as a mechanical device, for example.
- The size and scope of the system – clearly small personal devices require a very different approach to large plant level systems.
- The risk level of the system – factory machines have a very different risk level than nuclear power plants.

Thus, there are many variables and there are no generic answers and we should not copy the practices from another field blindly, but try to understand the context and see what possibilities agile approaches might give and in what way they should be applied – what parts of current practices they could replace, how they should be supplemented; are there “obvious” agile practices to implement and which agile practices definitely should not be used in the given context.

2 Goals

The goal of this report is to give guidance to companies that aim to change their processes to more agile way:

- How to apply agile principles is safety-critical software development.
- How to implement safety-critical design processes in agile based process.
- How to meet IEC 61508 standard series requirements in an agile project.
- Things to check when assessing the process.

The safety-criticality of the context of this paper is moderate. We think mostly of safety integrity levels SIL levels 1, 2 and 3 (see Safety Integrity Levels, Wikipedia Article) in this analysis. The methods for assigning SIL levels to a system are based on hazard/safety/risk analysis at system level and can be found in standards IEC 61508-5 and EN 62061.

The standard series IEC 61508 was selected as a reference for this analysis, because it is the most important basic safety standard for developing safety-critical software for machines and various automation systems. IEC provides a Frequently Asked Questions site for the standard series, which explains the standard's approach and application nicely (IEC 61508 FAQ). IEC 61508 is also considered to be quite challenging, so if the agile processes can be used with it, things should be easier than with many other standards.

Also, the standard series has been renewed in 2010 and this analysis provides a well-timed opportunity also to address some of its changes and their impacts on the development process and tasks.

The most relevant parts of the standard series for this analysis are:

- 61508-1, 2^{nd} ed., Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements. This, as the name implies, gives the general requirements for development.
- 61508-3, 2^{nd} ed., Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements. This contains the requirements for software development (with more elaboration of those in 61508-7).
- 61508-5, 2^{nd} ed., Functional safety of electrical/electronic/programmable electronic safety related systems – Part 5: Examples of methods for the determination of safety integrity levels. Determination of the safety integrity level is a very critical task, and this standard gives guidance for this.

3 Methods

The study uses the following main methods:

- Identification of agile principles, process and activity elements and practices that are key issues from the viewpoint of developing safety-critical software systems.
- Analysis of the identified elements: what possible risks might be in applying of them, how they should be supplemented, modified or avoided and by which means.
- Mapping of already identified required or otherwise essential tasks of development, quality and safety assurance into a generic agile development framework.
- Synthesis of key issues for guidance in process tailoring and development in companies.

4 Related research

The body of research on agile methods itself is vast, just like research on safety-critical systems and product design. Here, we will look briefly on the intersection of those areas. Cawley et al (2010) made a literature survey on applying agile methods in regulated environments and found only a small number of publications which, they think, “could indicate a very low-level of adoption of Lean/Agile methods in regulated, safety-critical domains, however, it may simply indicate a reluctance of companies in these domains to make their internal practices public”. In their paper, they report some issues and solutions to how to make agile work, and most importantly note the importance of organisational factors, including the need for management training.

Paige et al (2008) have studied agile in the development of high-integrity systems, including the analysis of elements in agile and the adaptation of agile processes. Their main finding was that agile methods can be adapted to safety-critical development by not replacing plan-driven processes, but applying them in appropriate tasks.

VanderLeest and Butler (2009) analyse the issues from the perspective of aerospace industry and present an analysis of agile practices and a mapping of agile process to standard DO-178B. They divide the agile practices into three classes: 1) fully compatible agile practices, 2) easily compatible agile practices and 3) problematic agile practices. Their results are not directly applicable due to the very specific requirement in aerospace development, so we are not going to go into detail here. As a conclusion they see that agile methods can be applied, but more co-operation is needed within the aerospace community.

Rottier and Rodrigues (2008) present a case study from the medical device industry. They found some problems in applying Scrum (see Wikipedia article Scrum (development)), a widely used agile process, including long validation cycles, strict regulatory standards and a high level of dependence on physical devices. The authors found a way to overcome the problems using, for example, test automation. The project was a pilot in their environment and thus did not yet bring increased efficiency compared to the old non-agile process.

Ge et al (2010) present an approach to how incremental methods can be used in safety-critical development. They claim that agile methods can provide benefits, but the methods are not directly applicable in regulated areas. They suggest up-front design in the process that at least produces information for a hazard analysis, before the agile portion of the process begins. The iterations of the software that the agile process produces also need to include sufficient arguments that the software releases are sufficiently safe. For large-scale development they propose a modular system where the modules are dependent on each other by arguments. Mostly the arguments seem to be in the form of safety requirements or safety goals to be met by the system.

Jacobsen & Norrgren (2008) in a master thesis present an interview-based case study of agile in companies producing medical technology. Their aim was to assess which agile principles would not fit into the software development in that domain and found agile to have many clashes.

Agile and plan-based methodologies are often understood to have natural areas or "home grounds". Boehm & Turner (2003a) present one analysis of this, but since the research is from a time when agile software development was a new phenomenon, we will not consider that research to have much confidence anymore (in 2011).

Besides the research mentioned, the applying of extreme programming (XP) has been researched in some studies, but because XP has been widely abandoned lately as a complete method – only its practices are used in the context of other methods – we will not look into that research here.

5 Requirements for a safety-critical software development process

Before we go into analysing agile development, we need to form basic reference criteria by which we reflect on and evaluate the agile world. It does not need to be complete, but just an outline of what necessary and is preferred for the process. (This work with principles will continue later as a process evaluation checklist, which organisations can use to assess their new process designs.) This author has outlined the following list of process requirements, based on the relevant safety standards and generic design knowledge. The list is not complete, but aims at this stage of the analysis mainly to be illustrative and to present a generic framework of “pre-understanding” of what we must look into when developing safety-critical systems using any kind of process:

Knowledge of risks

- The process shall utilise hazard and risk analysis of the target. If sufficient analyses are not available, they need to be carried out.
- The process shall include a thorough analysis of safety requirements. This requires a thorough analysis of the system’s actual usage.
- The process needs to share the safety requirements with the whole team so that everyone understands what is at stake.

Quality

- The level of quality assurance shall be high, at all abstraction levels of the product.
- Quality and professionalism of the process shall be high.
- As development of safety-critical systems is a task for experts, we need to expect good skills from all participants and also the process features can be chosen based on the skill level.

Control

- Due to the magnitude of tasks, the process needs professional management and control.
- As safety-critical development brings in more complexity, the process should aim for simplicity and clarity where it can.
- Collaboration.
- Safety engineering is an expertise field and safety engineers and analysts need to participate in the process.
- Safety requires good support for communication and teamwork.
- The need for required independence in validation needs to be possible without compromising collaboration.

Analysis

- The process needs to include natural places for hazard and risk analyses and safety assessments, at least at the level that the required standards require.
- Knowledge of the object of development.
- The process needs to include proper configuration management so that the system versions are fully defined when their safety is assessed.
- The safety information needs to be documented and the documents need to be updated when the system changes; at least when new versions are validated with the goal of deployment in any kind of customer environments.

Time and resources

- There needs to be sufficient time and resources to carry out these tasks properly, without the risk that compromises are made because of schedules. Therefore, safety tasks need to be taken off the critical path of the project.
- If the process produces changes to the product often, verification and validation needs to be as effective as possible, or/and moved off the critical path of the process. The same applies to documentation.
- All practices that are mandatory by standards or highly recommended, can be easily and reliably executed in the process.

Auditability

- The process must be auditable during and after execution.

6 Anatomy of agile development

When we consider agile development, we need to have an understanding of what it consists of. For the purposes of this study, agile software development consists of the following elements:

- Principles.
- Project model.
- Software development lifecycle.
- Software engineering methods, techniques and practices.

The principles consist of values, development principles and policies and thinking patterns of individuals and occupational groups and stakeholders. The project model is the concept of project planning and execution. A big part of the project's execution consists of developing software using a specific software development lifecycle, which in turn utilises various software engineering methods, techniques and practices, some of which may be developed especially for agile development, but some will have more generic origin.

Basic agile processes are by now understood to have "lost" some respected software engineering practices, and we need to be careful to analyse how these will not remain lost in the safety-critical context. Coplien and Bjørnvig (2010) outline those practices as being the following:

- Architecture.
- Handling dependencies between requirements.
- Foundations for usability.
- Documentation.
- Common sense, thinking and caring.

There have been some methodological approaches to bring some of the missing elements in place (for example Lean, which is analysed later in this report). But in the safety-critical context, the analytical elements have deep roots, which can likely address the deficiencies of agile, when applied properly. In all cases, the basic agile processes found in textbooks need to be supplemented with any practices that a given context or development situation requires. That is one thing that we are trying to do here – try to assess how common agile processes need to be modified in order to be appropriate in safety-critical development.

An agile development process is thus usually never "fully agile". The so called hybrid processes combine agile practices with traditional ways or doing things. Kennaley (2010) has analysed software development processes in a historical context and presents outlines for the next phase of software development, which again combines the best parts of various development paradigms.

In addition to the aforementioned elements, agile development is really not just a project execution paradigm or a type of engineering, but a culture, which means that when we are “going agile” we need to consider organisation culture issues, psychology and dynamics besides process issues. But those are not in the scope of this paper, except for an analysis of the agile values.

7 Applying agile principles in safety-critical development

The most important agile principles are at the time of writing (March 2011) the agile values expressed in the Agile Manifesto:

> *"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."*

We will start by shortly analysing the values in Table 1 for what they would mean in a safety-critical context and how they could be implemented. This is just a preliminary, rough level analysis. We will reflect on these issues many times in this report.

Table 1. Analysis of agile values defined in Agile Manifesto.

Value	Special meaning in a safety-critical development context	Practical principles to fulfil the meaning
[We value more] individuals and interactions [than] processes and tools	Substance and understanding and sharing safety information is of the utmost importance.	Safety information should be discussed and not just be read in documents and analysis reports.
[We value more] working software [than] compre-hensive documentation	True safety is more important than filling safety requirements (though the latter are mandatory).	We shall openly analyse safety and respond to real hazards first. Standards help in that, but we must not work only by standards. We need good systems, not systems that are documented as being good and safe.
[We value more] customer collaboration [than] contract negotiation	While safety issues and features are important, they are always things that need collaboration so that we can find practical, working solutions instead of non-robust ad-hoc solutions that cause more problems than they solve.	Active collaboration with customers on safety-critical features.
[We value more] responding to change [than] following a plan	When situations change, we need to assess the implication for safety immediately and not just blindly follow a project plan.	Every development increment must keep track of safety issues and respond to changes immediately.

What the values mean more in practice is explained in the twelve Principles behind the Agile Manifesto (http://www.agilemanifesto.org/principles, the practices and agile values are also explained by Cockburn, 2007). They are analysed in Table 2.

Table 2. Analysis of the twelve principles of agile development.

Principle	Special meaning in a safety-critical development context	Practical principles to fulfil the meaning
Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	Early releases shall be safe and able to provide value.	Safety of early releases needs to be validated; hazard and risk analysis, safety assessment and testing needs to be an on-going activity. This does not necessarily imply test automation, but an actively on-going testing activity.
Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	Safety requirements shall not hinder making sensible changes that provide value.	The product architecture needs to be flexible so as to encourage good changes that add value without compromising safety.
Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.	–	Periodical releases shall not be compromised in their safety.
Business people and developers must work together daily throughout the project.	People who are responsible for safety or who are responsible for assuring it should work together with the development team.	Hazard and risk analysis and safety assessment should to be a team effort, led by independent professionals.
Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	People who participate in the development must be motivated by safety too and they need to be given resources and tools to use that motivation.	Gradually, safety and reliability analysis tasks can be given to the development teams' tasks, yet independent analysis can be required to be carried out by an independent party.
The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	Safety information is shared in face-face meetings and not just assessment reports. Safety issues need to be given a place in meeting agendas.	Participation in risk analysis is a very good way of sharing information, yet the analysis still needs to be documented properly.
Working software is the primary measure of progress.	True safety of the software is one measure of the progress; not just how designs and plans pass safety assessments.	One metric of progress is how efficiently the process can deliver good, working software that is also safe to use and meets the safety requirements. Safety requirements need to be based on risk and safety assessment, not just standards.

Principle	Special meaning in a safety-critical development context	Practical principles to fulfil the meaning
Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	Working overtime exhausts developers and testers and makes them overlook essential factors. Tired people should not participate in safety-critical development any more than they should be exposed to risks in using machinery. Developing of safe systems also benefits from experience, so it is not good if developers change jobs due to exhaustion.	Work on safety features needs to be planned and resourced realistically, just like any other development activity.
Continuous attention to technical excellence and good design enhances agility.	Safety, properly implemented, is technical excellence. Safety features need to be properly designed, not just add-ons. There must be absolutely no design flaws in safety systems.	Work on solid safety architectures and elegant integration of safety features to the general architecture is essential.
Simplicity–the art of maximising the amount of work not done–is essential.	Simplicity and understandability of safety-critical features are essential qualities. Simple safety features are easy to adjust to the changing functionality.	Simple base architectures for safety features should be designed.
The best architectures, requirements, and designs emerge from self-organising teams.	The team should have freedom for the design of safety features, but not safety requirements. Yet all decisions need to be bases on solid analysis and proven (or provable) techniques. The person who is appointed to be responsible for safety issues has the final vote on all decisions, whether she/he is part of a team or not.	Present the safety requirements to the team clearly, let them understand what they mean and what their implications are and let the team do the designing as it best fits the whole system. Yet the results need to be validated in a sufficiently independent way.
At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.	How well development of safety features has succeeded needs to be assessed as part of the team’s self-assessment.	Reaching of safety goals as part of lessons learned – agendas and such.

8 Implementing agile processes in safety-critical development

8.1 A generic agile development process

For the basis of the analysis we first outline a generic, simple model of agile development that presents its most prevalent features. We do not want to use a specific process model such as Scrum as basis of this analysis, as specific models tie our analysis unnecessarily to their highly specialised details – and because in any case the specific process should preferably always be tailored to specific circumstances anyway.

Our simplified model (see Figure 1) has the following elements:

- Process flow.
- Start-up activities. Pre-development tasks that are done before the increments – concepting work, etc.
- A series of increments, adding more features or otherwise more value. (Note: in some processes these recurring process phases are referred to as "iteration". We use the term increment here to avoid confusion. Inside the increments there will be plenty of iteration, as designs evolve through their analysis and we must not confuse that with the process phases.)
- A rhythmic series of releases, at the end of one or more increments. Not all increments need to produce releases and thus the need for results of the increments to be safe, or validated to be safe, varies.
- The releasing increments produce releases directly to the customer or production, or release candidates for additional internal processing (to be passed through required, for example, product management processes).
- Closure activities. Post-development tasks that are done after the increments.
- Ongoing activities, such as testing and safety tasks that are carried outside the increment model.
- Management and control processes that are outside the series of increments.
- Practices, related to, for example, integration and testing.

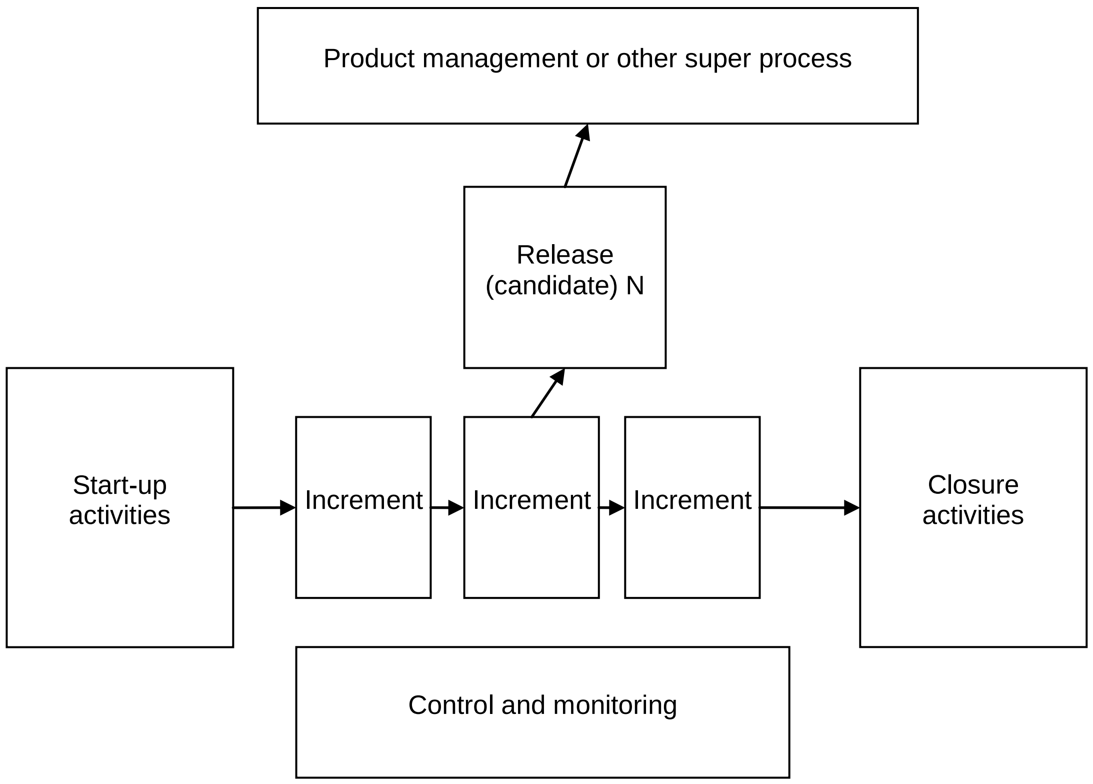

Figure 1. The basic agile process.

8.2 The "home ground" of agile

People usually have thoughts about in which kind of environments agile development might be "at home" and where not. This is an evolving issue as more and more is learned of agile development. For orientation we present the view of Boehm and Turner (2003a), but with a word or caution: the table should not be read too literally as eight years have already passed since it creation, during which time more have been learned of agile.

Table 3. Agile and plan-driven method home-grounds (Boehm and Turner, 2003a and with detailed explanations, Boehm and Turner, 2003b).

Characteristics	**Agile**	**Disciplined**
Application		
Primary Goals	Rapid value; responding to change	Predictability, stability, high assurance
Size	Smaller teams and projects	Larger teams and projects
Environment	Turbulent; high change; project-focused	Stable; low-change; project/organization focused
Management		
Customer Relations	Dedicated on-site customers; focused on prioritized increments	As-needed customer interactions; focused on contract provisions
Planning and Control	Internalized plans; qualitative control	Documented plans, quantitative control
Communications	Tacit interpersonal knowledge	Explicit documented knowledge
Technical		
Requirements	Prioritized informal stories and test cases; undergoing unforeseeable change	Formalized project, capability, interface, quality, foreseeable evolution requirements
Development	Simple design; short increment; refactoring assumed inexpensive	Extensive design; longer increments; refactoring assumed expensive
Test	Executable test cases define requirements, testing	Documented test plans and procedures
Personnel		
Customers	Dedicated, collocated CRACK* performers	CRACK* performers, not always collocated
Developers	At least 30% full-time Cockburn level 2 and 3 experts; no Level 1B or -1 personnel**	50% Cockburn Level 2 and 3s early; 10% throughout; 30% Level 1B's workable; no Level -1s**
Culture	Comfort and empowerment via many degrees of freedom (thriving on chaos)	Comfort and empowerment via framework of policies and procedures (thriving on order)

* Collaborative, Representative, Authorized, Committed, Knowledgeable

** These numbers will particularly vary with the complexity of the application

(Note: Boehm and Turner (2003b) also explain the levels as: 3: Able to revise a method (break its rule) to fit an unprecedented new situation; 2: Able to tailor a methods to fit a precedented new situation; 1A: With training, able to perform discretionary method steps; 1B: With training, able to perform procedural method steps (...); -1: May have technical skills, but unable or unwilling to collaborate or follow shared methods.)

8.3 Analysis of process elements and practices

8.3.1 Process feature: Understanding of software concept

The concept is formulated in start-up activities, when someone – a product manager, a customer or some contract decides the main features of the software to be developed:

- What purpose is it developed for?
- Who uses it?
- What kind of rough workflow or business process it should implement?
- In what environment?
- What benefits should it give?
- What kind of risks might it have? (The general risk level of the system.)
- What is the general approach to technology?

This could also be called a vision phase. It should give everyone a shared vision of what the team should develop and deliver. After that phase, the gathering of requirements can begin. Visualisations and non-working prototypes are used to share the vision.

In agile development, this phase is important for team building, as a good team is necessary for the success of agile. How the team understands the goals is an essential factor in that. When the vision is discussed, people get to know each other better, they understand what kind of skills and people the team should have and what kind of external validation and verification processes and services might be needed.

8.3.2 Process feature: Product management

Agile development suggests that a business representative works daily with the development team. Even though the team might be self-directing, the role of the business representative is to represent the business side of things:

- Commercial and market issues.
- Customer related knowledge (if customers cannot be included in the development).
- Technology management issues.
- Product line level issues.

This representative is often called product owner and is seen as a virtual role to be filled by more than one person as required. Ownership here means in practice "taking responsibility on behalf of the manufacturer" and that is not something to be taken lightly, and we must ensure that all project role terms match the responsibilities.

A common division is a division into a business related and a technical product owner. In a situation where safety is a critical factor, a safety related ownership can be defined. Thus, whereas the business project owner defines the business objectives and general requirements, the safety project owner defines the safety objectives and accepts designs and implementations for their safety performance.

8.3.3 Process feature: Requirements specification and management

Agile development has usually stopped using traditional requirement specification and presentation techniques and started using user stories even to replace use cases. The viewpoint of user stories is subjective, and not sufficient, as most safety requirements are objective and contain standard-defined design and implementation requirements. So the requirement management process cannot rely on agile culture, but needs to utilise the traditional techniques.

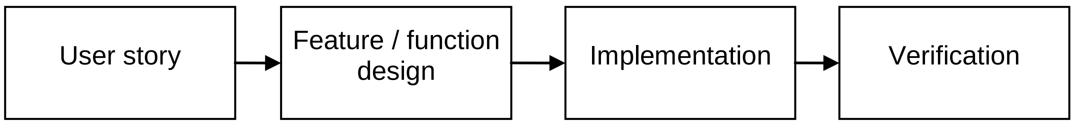

Figure 2. Basic "requirement specification" in usual agile practice.

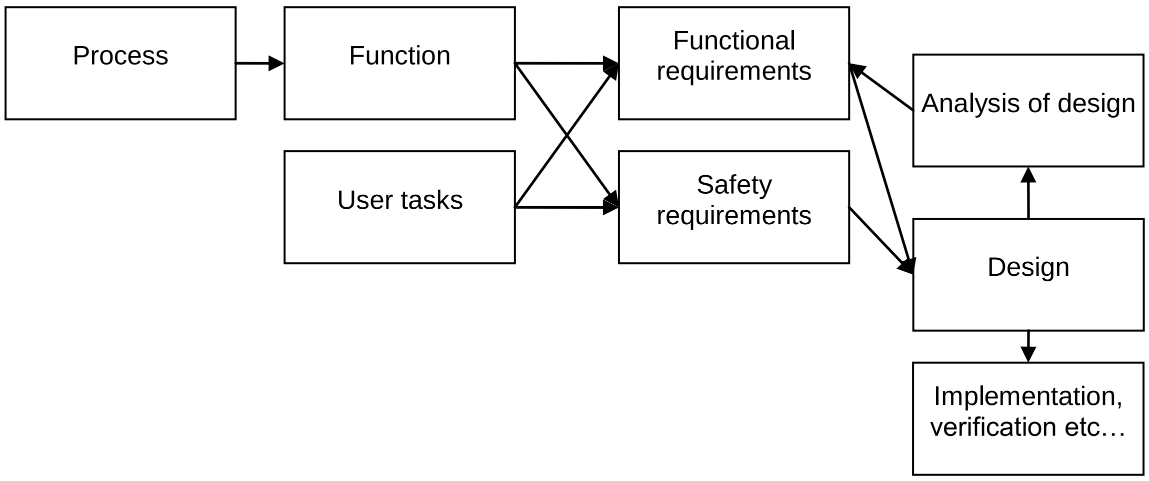

Figure 3. Safety-critical development requires a richer approach even in an agile process.

User stories describe the work of operators and other actors in the work system, but do not do that in a systematic way. For safety-critical systems, hazard and risk analysis that study users' actions can be a tool for systematising descriptions and to gather safety requirements that represent the true usage. This is a very agile principle.

System level risk analysis is an obvious task at the start-up phase. With that we find out both threats to the system, users, business and society, and determine the safety and reliability levels that development should work within, and reach, in the delivered system.

8.3.4 Process feature: Release plan

All incremental process models have some form of a release plan. Even in agile development there is a rough idea of what kind of features should be delivered to the customer during the development project. The main difference to some other models is that the idea should be really rough and there should be no commitment to any specific features – during the course of the project it will be shown what features will actually be developed and implemented.

The release plan is sometimes called a road map, which usually means a systems lifecycle spanning several projects and it can traditionally be technology-centred.

For the release plan to be meaningful it requires good (which does not mean heavy) concept design before the project really starts.

For safety-critical development this project level roadmap brings important benefits:

- It enhances the shared vision of the system and increases the knowledge of all participants.
- It softens any sudden changes in development plans, due to learning.
- It gives some view as to what kind of safety systems might be needed.
- And finally: it gives an estimate to points in the process where validation and certification will take place. As those are time-consuming efforts, they really need to be thought about carefully.

So, in safety-critical development the release plan and validation plan need to be developed together.

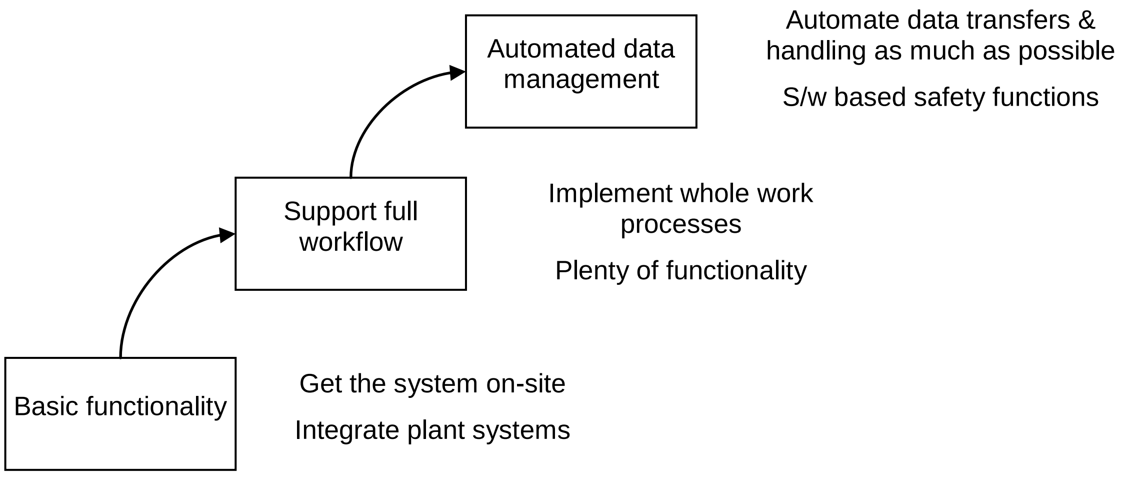
Automated data management
Automate data transfers & handling as much as possible
S/w based safety functions
Support full workflow
Implement whole work processes
Plenty of functionality
Basic functionality
Get the system on-site
Integrate plant systems

Figure 4. A rough release plan in safety-critical development (illustrative).

8.3.5 Process feature: Usability design and usability assurance

It is now widely accepted that other safety related non-functional requirements are not properly addressed in agile development. Paradoxically, one of these is usability. Usability has direct effects on safety, as it aims at reducing human errors (usability is not just about making usage easier). Agile aims at direct communication with users and letting their "voice be heard". Yet, good usability design and assurance requires special skills – both in development and in assurance. Tackling usability issues can be done in many process phases:

- The concept design phase outlines the mode and patterns of use.
- Design of new features requires analysis of work, preferably including work related hazard and risk analysis.
- Design of safe user interfaces requires good designer skills and knowledge of the usual UI types in the particular context and industry.
- Evaluation of UI design and implementation requires usability analysis skills and skills on running user tests (on implementation or on prototypes).

This analysis-evaluation loop forms a natural feedback loop for the team.

In safety-critical development usability is closely linked with safety. Safety assessment of new or updated user interfaces should include systematic analysis of the possibility of human errors.

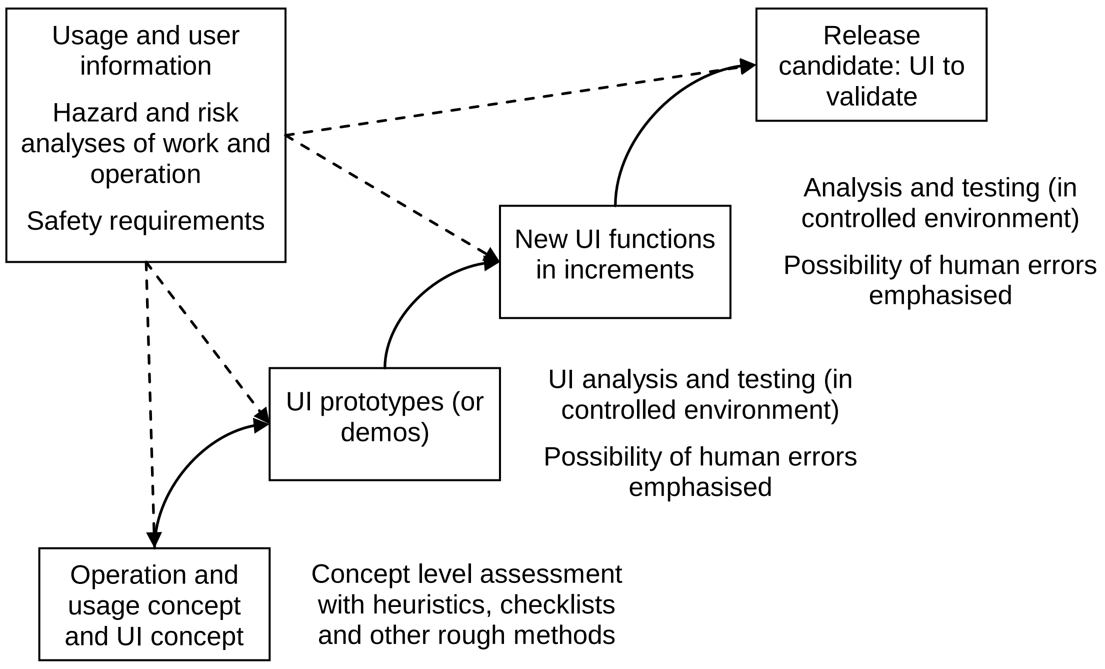
Usage and user information
Hazard and risk analyses of work and operation
Safety requirements
Release candidate: UI to validate
Analysis and testing (in controlled environment)
Possibility of human errors emphasised
New UI functions in increments
UI prototypes (or demos)
UI analysis and testing (in controlled environment)
Possibility of human errors emphasised
Operation and usage concept and UI concept
Concept level assessment with heuristics, checklists and other rough methods

Figure 5. Usability assessments and testing during development.

There are some possible benefits of agile processes to user interfaces:

- UI development can be more tightly integrated with the rest of the system development, using a tight iteration loop. This has good potential for better, safer user interfaces.
- Updated user interfaces are implemented in every increment. Thus there are good "points in time" available for their assessment – both for generic usability, and for safety.
- It is easier to concentrate on UI details when development is carried out in smaller parts.

But still, the same issues can cause problems too:

- Usability requires good up-front planning so that a proper user interface concept is developed and chosen.
- When changes are made to the user interface during the course of the project, it will get worse, unless it is properly redesigned at some point(s). A "refactoring" of details is not sufficient.

So, as long as professional UI design and usability assurance practices are applied, agile can bring benefits to usability and help create more robust and safe user interfaces and usage patterns.

8.3.6 Process feature: Architecture design

Design of architecture is one critical phase of any development project and more so in a safety-critical context as the safety features need to have a solid relation to the functional architecture. A common criticism towards agile is that the architecture is often neglected and when it just evolves it does not come out as well as it should. One problem is that traditional architecture design description is too exact and too close to implementation and thus does not support change, which is essential in agile development.

However, in safety-critical development, the architecture should be upfront-oriented: in the concept phase, before the increments, a generic form of the architecture should be formed and developed – but not too far – during the first increments, in collaboration with the whole team and other participants.

In safety-critical development, this issue is complicated due to the need for differentiating different kinds of architectures. While SFS-EN 61508-4 defines “architecture” as being “specific configuration of hardware and software elements in a system”, the reality is more complicated. The following division of architecture is most relevant:

- The functional architecture. The architecture of the functional software system than implements the systems functions and business processes. The functions may or may not be safety-critical. Mostly, elements of the functional architecture may cause hazard due to their malfunctioning because of failures, design and implementation errors, human error in operation, and improper configuration among other reasons.
- The safety system architecture. Architecture of the systems that provide monitoring systems to the functional system; safety devices and other system elements which assure the system’s safety both in normal and abnormal situations.

Both architecture types need to be properly designed and they also need to be independent of each other. The independence is most important in implementation so that, for example, when the functional system has a failure due to environmental influence, hardware or software problem, the safety architecture elements are not affected and will continue to work as planned. It may even be beneficial if the safety architecture is based on different philosophy and different architectural patterns so that it is not affected by the same root causes or problems as the functional architecture. (For example, a communications problem caused by a certain design solution in a certain situation should not cause problems in the safety system's communications capability.)

This clearly requires high-class specialist expertise and good, sound principles laid out early in the process.

Another issue is the diversity of designs, required on higher SIL levels. It means that redundant safety functions should be developed with varying technologies so as to avoid common cause failures. This means that a safety function might be required to be designed in two or three different ways. The ability to do this will require existing, reusable solutions and again an architecture that is not tied to implementing technology.

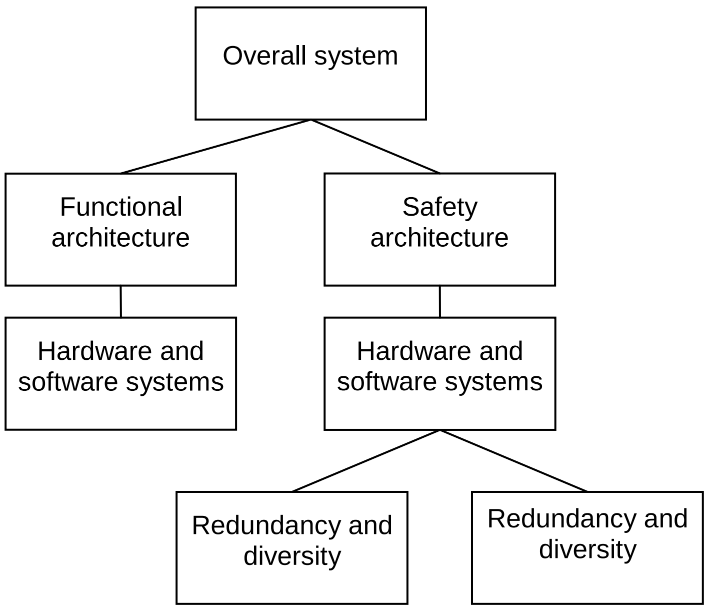

Figure 6. Levels of architecture in safety-critical systems.

8.3.7 Process feature: Reliance on increments and timeboxing

Agile development relies on development steps, called increments, iterations or sprints. Note that an increment can be used in two meanings: an increment of the software product or the development step that the new version of software is created in. In this report we use the term for the process steps instead on iterations, because iteration would sometimes imply reassessing and reworking of previous design and development, which does not necessarily happen.

The idea is that all development tasks for new features are carried out during an increment. The idea is to select just so many features for development that the team can manage the completion of the tasks with high quality and satisfaction. This principle is also called timeboxing and it is in stark contrast to traditional V-model development.

However, not all the validation and verification tasks of a feature or the whole software system can be performed or are sensible to carry out during a two-week or 30 day increment. This depends on the type of increment: if the increment aims at a version to be tested in a simulator or an internal highly controlled test machine installation, all tasks necessary for that goal can be carried out.

But if the increment aims at a release to production work or to be tested by customers, some more time may be needed. In that case, the result of the increment needs to be frozen (that is, no changes will be made to the code or configuration) and necessary tasks for safe delivery and deployment can be carried out in parallel during the new development increments. These include:

- Impact analysis.
- Updating of safety and risk analysis based on new development.
- Regression testing.
- Independent validation testing.
- Assessing conformance with requirements.
- Internal acceptance.
- Delivery to customer.
- Customer acceptance testing.
- Deployment in production or customer's test environment.

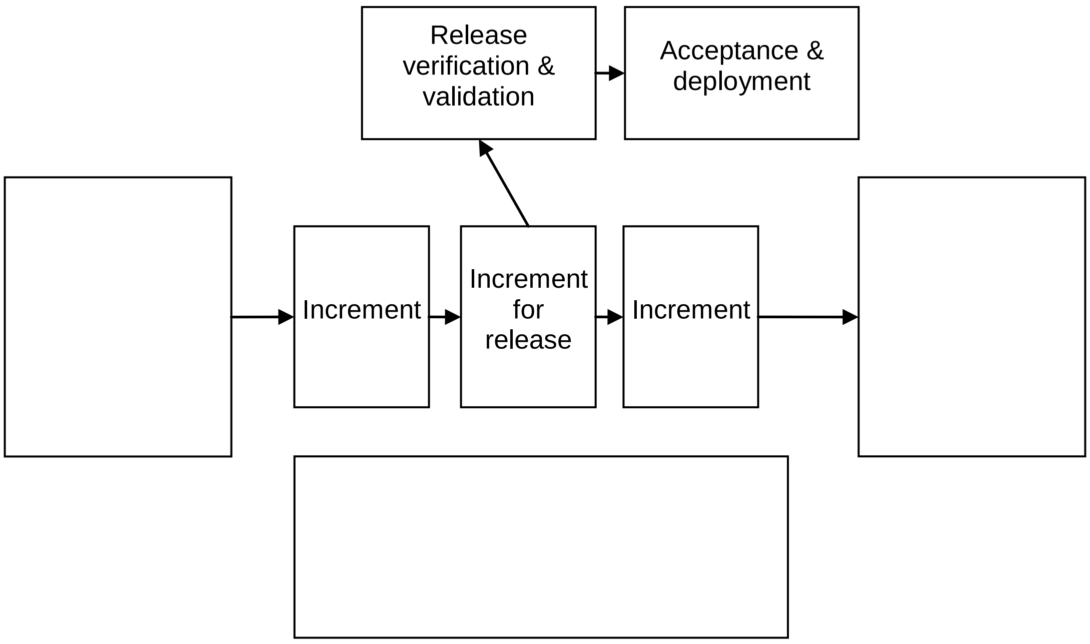

Figure 7. The parallel release process.

One important consideration is how we understand the role of the new versions to be released. Are they already “releases” or just “release candidates”? When we call them “candidates”, we emphasise understanding that the new version still needs to be validated in a “super-process” of the core software process. The super process will have a safety validation and acceptance function, but it also may need to be assessed from sales and product management perspectives, meaning that it is subject to business level decisions.

The release also needs various other activities other than validation or any other software engineering process. They may include product level, documenting, internal and external training, contract and legislation related tasks etc.

The length of increments is an important issue to think about. It can vary between companies and projects from one week to three months (in any single project all increments should of the same length). The most common seems to be 30 days. It should be expected that safety-critical development would favour longer increments than other kind of development, due to the number of tasks each developed feature requires, but this is something that needs to be thought about case by case. Of course, with longer increments the process might lose its agility and might be more like a traditional multi-release process.

8.3.8 Process feature: Increments as process flow

One way to look at the flow of increments is to see them as individual processes, where increment N produces outputs to increment N+1, and increment N+1 has formal inputs from increment N. This is a traditional process pattern that may help us structure the usage of safety related information

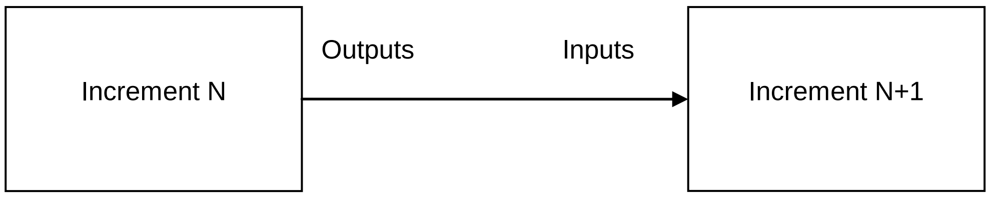

Figure 8. The increments seen as a process flow.

Thus, each increment should have a stated requirement to produce the necessary outputs to the next increment, including:

- A new version of the software.
- A specified new configuration of the software.
- Hazard and risk analysis and safety assessment of new features.
- Verification and validation information of the new features, including results of regression testing.
- Updated hazard and risk analysis of the whole software system.
- Updated required documentation.
- Information of any other change.

- Internal and external assessments of the new software version and other products.
- Project and process information.
- Updated list of remaining development tasks that are known.

The next increment in turn uses those as inputs, but also utilises other information, including:

- Changes to the total system (including hardware and environment).
- Changes to customer or market needs.
- Information on available new technology or other applicable developments.

And the first step of the next increment is to assess the inputs and start planning new development tasks based on that analysis.

8.3.9 Process approach: Risk-based development

Agile is by nature a risk-based approach. One central idea is to assess which features of the system provide most value to the customer and then develop those before starting to develop other features. This should mean that the most important features receive proper, focused attention, good design and verification and validation. A by-product of this should be a safety architecture that best supports the most important use cases and processes that the system will execute and be used in. Because the focus is on small parts of the system at one time, the safety, risk and reliability analyses made should of better quality than if a larger specification or design was being analysed.

Of course, the features and functions that provide the most productive values to the customer may not be the ones that are more hazardous. Therefore the safety design side must make sure that “secondary” tasks, such as maintenance and the handling of inevitable disturbances receive proper attention. That is because exceptional situations traditionally cause the most risks. A proper hazard and risk analysis will bring these issues up, but in agile there is a risk that they may be masked by too lightly described and analysed usage / operating scenarios.

In fact, the priorities for development need to be defined using multiple criteria, for example:

- Value for the customer in production sense.
- Hazards and risks involved.
- How important and defining the functions are for the functional architecture and systems.
- How important and defining the functions are for the safety architecture and systems.
- How much the developed item helps all participants in understanding the system and aids in learning.
- How well the issues related to the item suggested for development are understood. In general development, functionalities can be developed just to see how they work, but for safety-critical development, all issues should be understood better, so as not to have any surprises. The various analysis methods are just for this.

In agile development this prioritisation and selection of features for development happens at the start of each increment and it needs to be ensured that it is done in a risk-conscious way. This is a phase that is also critical for sharing risk information within the team and stakeholders.

All in all, this is an aspect where agile development could provide essential benefits compared to traditional processes.

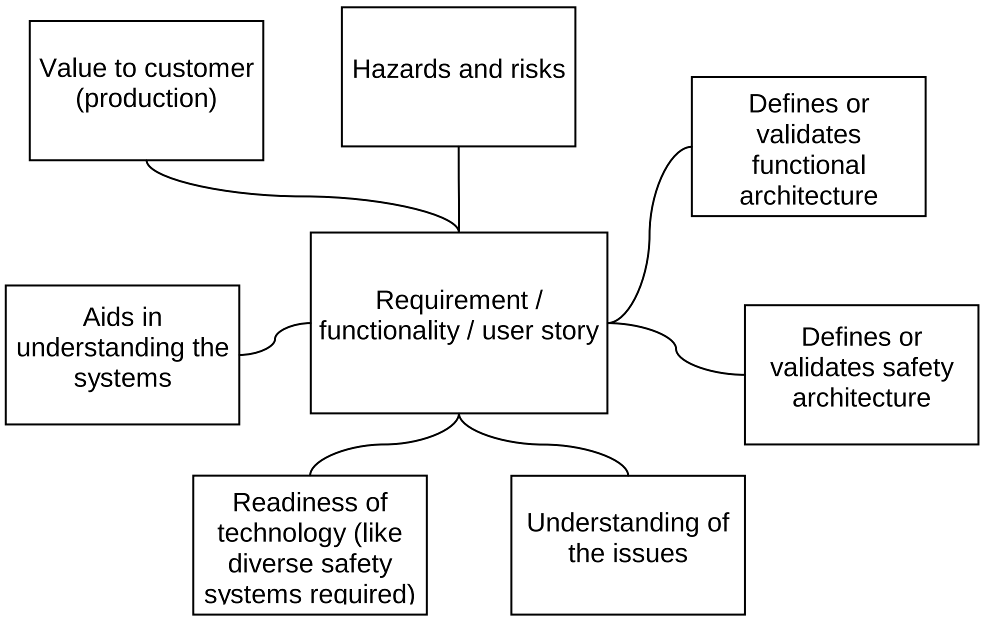
Value to customer (production)
Hazards and risks
Defines or validates functional architecture
Aids in understanding the systems
Requirement / functionality / user story
Defines or validates safety architecture
Readiness of technology (like diverse safety systems required)
Understanding of the issues

Figure 9. Some issues related to a requirement.

8.3.10 Process feature: Configuration management

Another increment-related principle is that the product should be sufficiently mature for use after an increment. This means that the product (and the configuration) includes only functionality that has been properly tested and found to be suitable for use, or other purposes of the increment. If it happens that the development of a feature which had been selected to be developed during an increment, meets problems and cannot be finalised, in agile development that feature can be left out of the product. However, should that feature be safety-critical or should its absence compromise the safety, this clearly cannot be done. Thus, the selection of features to be designed and overall planning of the next increment need more consideration in safety-critical development.

Configuration management in agile processes is usually based on managing versions of software modules. In safety-critical development, the configuration includes a lot more product related information, especially safety information, attached to each functional configuration element. The auditability and reporting functions of configuration are thus important in safety-critical development. The selection of configuration management systems should include criteria for handling documents, besides software code and objects. For example, some configuration management systems cannot report differences in documents, unless they are in textual, non-binary form or some managed structured form.

8.3.11 Organisational feature: The development team

A “team” by definition means a group of people who together decide on their work: who does what and how; how they collaborate etc… This type of team is called self-organising. A traditional project group is not a true team, as the project manager usually decides the division of work and work methods.

Teams that also decide what they should do, are called self-directed teams. That is possible when all system “owners” participate in the team, but it is questionable, how often such a team could be a true team in system development context, other than in low level technology development, under strict conditions (for example, component development to do some specified task or free R&D style concept development, the results of which will not necessarily be used commercially, but provide alternative models for the future).

Teams can function that way when the following requirements are met:

- The development goal is fully understood.
- When the team can represent all necessary stakeholders sufficiently.
- The team members each share a wide and versatile skill-set for dynamic work load division.
- All the necessary special skills exist in the team.
- The team has undergone a teaming process. This is playfully said to consist of forming, storming, norming and performing phases, see Rowley & Lange (2007) for a description of the phases and case study that describes many of the phenomena associated with the process. Scharmer (2001) presents another view to the process from a viewpoint of how language is used in dialogue within an organisation. In a multi-disciplinary team, use of language is very important issue, but one which we cannot go any further in this report.

Some of the first application of self-organising teams were first used in an industrial setting in the automotive industry (in Sweden and soon also in Finland) to replace production line work with work shells, where a team could by themselves divide assembly tasks in suitable way; each helping each other. Another area was R&D type product development, where the team was given an opportunity to find new concepts. It is also important that the team has gone through a team building process and can function as a team. (For this, a team-builder’s profession seems to have raised its profile lately – companies cannot afford the team’s natural processes to produce a working team, but need external help to start, catalyse and guide the process, and to train personnel in the necessary skills of development teamwork.)

Safety-critical development has, however important differences:

- Safety-critical development requires many special expert skills, making it impossible to let the team divide their work in any meaningful way that would replace pre-planned ideas of what each member's role is.
- What the team should design and implement results from the customer's or product manager's decisions.
- In most development contexts, there are high schedule pressures and there is no time for the team to learn by making mistakes or traversing alternative development paths.
- Standards often dictate how the process should go and that implies who is in charge of what task.
- The whole product development activity requires far more "management" in the sense of control and coordination than, for example, development of a game or an internet site platform.

Even in other areas, the results from self-directing teams are not as good as what companies expected when they were more in fashion in the 1990's (Wilson & Whittington, 2001, refer to some of the issues) whether in the factory floor or in product development. In fact, almost 20 years ago, Nonaka and Takeuchi (1995) noted for automotive product development, that:

"The rugby style, however, has its drawbacks. Since this approach entails problem solving by an interdepartmental pool of personnel who share the same space and time, the process is liable to give too much importance to preserving overall unity and conformance. In other words, it may lead to the risk of achieving a compromise or consensus around the lowest common denominator. Since the relative influence of the manufacturing and marketing departments is strong, the rugby approach hinders a relentless quest of technological potential."

Safety-critical development should not settle for the lowest common denominators. Pure self-directed teams may not be without problems, but still agile teamwork principles have a lot to offer for team functions in safety-critical development:

- Discussion of features. Instead of a requirement analyst forming functional requirements, the team can, together with the customer, plan and develop these at least partly (conference room meetings need to be supplemented with proper studies at customers' sites, at factory floors, using interviews and observations etc.)
- Inclusion of team in analyses. The team can participate in safety and reliability analysis, lead by the team's safety engineer. Collaboration in the common FMEA/FMECA analysis, used in many companies, is the first step in this.

A very important feature of the team is that of a forum for stakeholders and external experts. Stakeholders' daily participation in the team is often recommended and can be practical in some cases, but not realistic in others. But it outlines a generic principle: in a waterfall development process, the team's representative (a designer, for example) goes and meets people and brings back information to the team. Now, it is expected that people come to meet the whole team to discuss things. Therefore, it is psychologically possible for a customer's safety, quality or maintenance person to come to meet the team and discuss all open issues openly with everyone. It can happen physically or in a video conference. This would be something to develop further.

The teams are often not lead by a “manager”, but a team leader, called, for example, in the Scrum method a “Scrum master”. A project manager may not even exist as a role in a project. Safety-critical projects need a lot more decisions and control and coordination with other parties and there is no evidence that a traditional project manager could be replaced with another role. Still, more analysis of roles in the development process may show new solutions here.

Still, independent of the management or leadership models, the teams need various kinds of guidance, and portions of each can be internal and external:

- Managing the development process itself.
- Quality management and quality leadership.
- Safety management. This is of the biggest interest in this research.
- Personnel management, including enabling of motivational factors and team performance.
- Product management.
- Knowledge management.

The term management does not imply a mechanistic “ordering” of tasks to people, but any activity that suits the company culture and ensures that the team works properly. Thus, it includes ensuring that all the needed tasks are carried out, shared thinking within the team is lead positively, the team has a good working environment (in the Nordic sense) and the working days of all the participants are satisfactory and productive. Continuous team building, catalysing communication and other “soft” management tasks form an important part of that job.

Sometimes, when it is assumed that the teams can do without management, they still receive plenty of external leadership an do not perform purely on their own resources. The leadership elements include the management showing good example, enthusiasm and moral; and not sweeping problems under a rug, but solving them promptly.

One special issue in leadership is forming a shared will to create safe systems. A "safety first" atmosphere is essential, shared by all from top management to all team members. This is the core element of safety culture and may also be the most critical factor for any form of team independence. For more on this issue, see Scharmer (2001).

Good leadership is one key issue to improving the teams' performance and perhaps gradual growing independence.

All in all, in safety-critical development, the following can be seen to be a selection of important issues for ensuring a team's performance:

- Forming a shared will of creating systems that are safe.
- Training and building of safety-related knowledge and competence.
- A mature quality culture and high level of professionalism.
- Stability of the team to overcome team dynamics and to build a mature team that can collaborate effectively.
- Leadership of the management: values, morals, example, problem solving; taking care of people. Commitment to the teams. Development of organisational culture to provide a good "climate" for team work.

- Psychology and role based team compositions. For a team to perform, its members must not be selected based on skills only.
- The team members' commitment and trust to teamwork.
- Training of teamwork.
- Training or the approaches, goals and languages of the various participating personnel groups and company functions. For example, if sales people and developers do not understand each other, collaboration can be difficult. Training can give a jump-start to the understanding that will grow during collaboration.
- Wage systems may need to be modified to support teamwork.

The issues underline, that the issue of teams is not just a methodological matter, but greatly related to organisational culture, and development of that is not a short-term task but long-term effort. Wilson & Whittington (2001) has a telling title: "Implementation of Self-Managed Teams in Manufacturing: More of a Marathon than a Sprint". Indeed, organisational behaviour and culture cannot be transformed in a lasting way by just implementing a new project management method. And obviously, they shouldn't, unless there is a good, demonstrated reason for it.

8.3.12 Process feature: Frequent meetings

Most agile processes emphasise frequent meetings of all participants. Scrum especially is known for its daily meetings. The idea is to have a meeting between people where people can not only give status reports, but share thoughts and help each other to meet their common goal. Of course, in distributed development, this can be challenging. Woodward et al (1010) have published a nice book about collaboration and communication in distributed development (while the book's title mentions Scrum, it is applicable to any kind of project management).

This is in strong contrast to traditional meeting practices, where the meetings are not as frequent and participants are chosen on need-to -basis. Many teams, especially from subcontractors, are easily left out of the meeting systems and need to rely on emails, documents and individual phone calls for documentation.

This open meeting culture should provide a lot of benefits for safety-critical development as it opens up the communications between all parties.

8.3.13 Process feature: Agile as a learning process, continuous self-reflection

Every software development process is a learning process. What we think of the system under development evolves. Gradually, we understand the process better. In traditional development, learning is a front-weighed process: by up-front analysis and prototypes we aim to understand the system as much as possible before starting to develop it. In agile, we learn by doing, and mistakes are allowed. When we make mistakes, we just adjust in the next increment.

This causes compromises. All features and quality factors cannot be in top shape from the beginning. When an agile process produces value in the beginning of the project, most often working features, it can happen at the cost of, for example, usability and performance. But in safety-critical development, every release to the customer must fulfil safety requirements. This will cause adjustment in the developers’ mindset.

Still, an essential element of agile development is the team's continuous self-reflection. While in more mechanistic processes that is an "extra" feature of good teams, in agile it is a very critical factor, because it is the internal feedback mechanism of the team. Cockburn (2007) includes a nice description of this. Most notable part of reflection are the feedback meetings at the end of the increments, but reflection should be more of a mindset and an all-encompassing aspect of everyday activities.

Safety-critical development presents some aspects that support reflection:

- Often repeated safety assessments in the form of, for example, FMEA analysis or with other failure analysis method.
- Safety represents reflective, important criteria to assess all functionality.
- Experts, who have varying viewpoints, collaborate in the development.

These are all naturally amplified if development is done in an agile way.

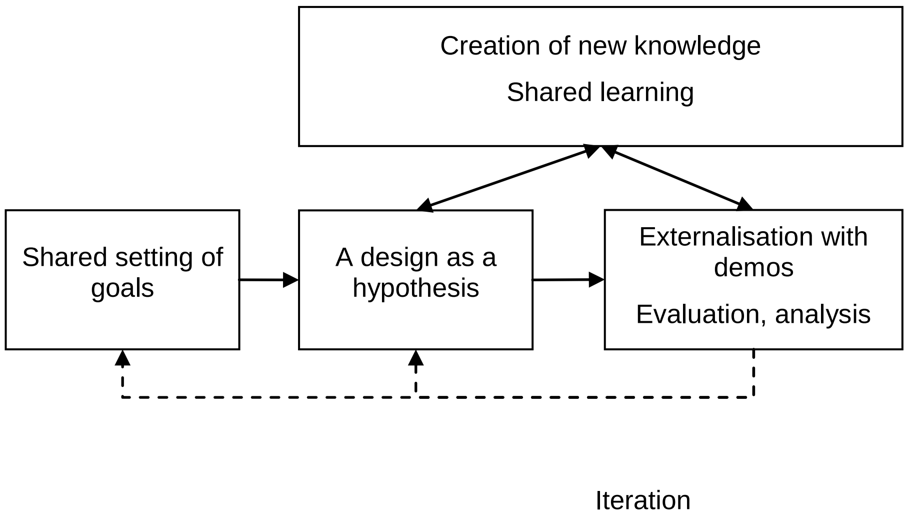

Figure 10. Development work as a reflective process.

8.3.14 Process practice: Small task-based development and included documentation and risk analysis

Usually, development tasks are planned at the start of an increment using a list of things to do, often called a "backlog". Development tasks are expressed as a goal to design and implement a small part of the system – a feature, enabling a use case to be performed, support for a user story or such like.

In the beginning of agile history, a task was often claimed as ready, when it was implemented, but now it is understood that a task is not ready before it has been properly tested, using test approaches above low level integration (module integration) testing.

This forms an important principle: any development task is not “ready” until all necessary team tasks have been carried out. In safety-critical development these include:

- Analysis of the effects of the new implementation to the system.
- Analysis of safety requirements and documentation of these.
- Design and implementation.
- Review of design and implementation.
- Hazard and risk analysis and safety assessment of the new implementation.
- Update of the safety documentation.
- Verification of the new implementation.
- Regression analysis and testing.
- Validation of the new implementation.

This may sound like a lot to do, but when the work is divided into each small development task, it can become as natural as unit testing.

Where the tracking of tasks progress has traditionally included design, implementation and testing of features, the list of subtasks now needs to be expanded.

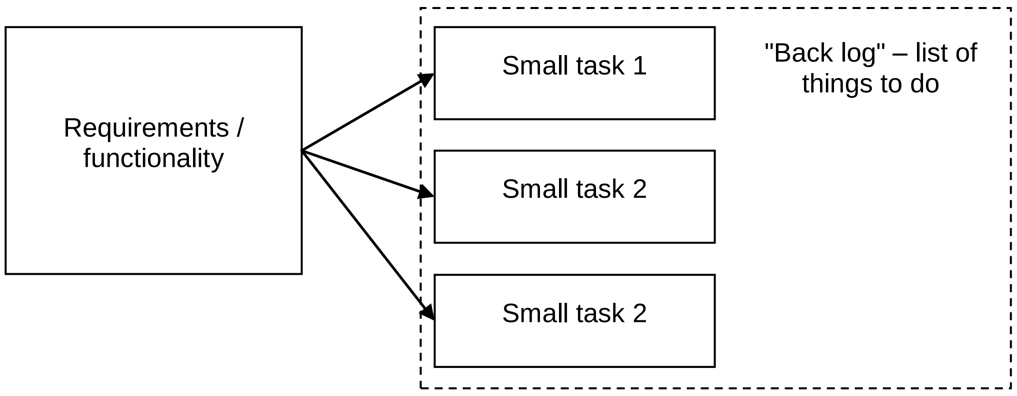
Requirements / functionality
Small task 1
Small task 2
Small task 2
"Back log" – list of things to do

Figure 11. Requirements split into small tasks to execute and monitor.

The small tasks are optimally of similar "size", meaning that they require a similar amount of work and thus allow for estimation of what number of such tasks the team can handle during each increment (called "velocity").

For safety-critical development such splitting and harmonising can have adverse effects, as safety requires a holistic view of the designs. So, this is one area where one should be careful.

8.3.15 Process feature: Documentation and analysis

Overall, agile culture is reflective in its working style. Developments are made and assessed and then the process is adjusted based on the reflections. This style applies in all elements of the process – in analysis of the requirements and in analysis of the resulting new designs and implementations. We will look more into the analytical process in appropriate places of this report.

Because of this, documentation is somewhat neglected. There has even been a misconception that in agile development, documentation is not needed – or is even forbidden – and it is replaced by verbal communication. For example, when in traditional development a chart has been drawn, it would be included in multipage documents and stored carefully. Some agile practitioners would draw the chart in rough form on a whiteboard, discuss it with the team and then discard it, which is indeed a very efficient way to communicate. But used in this way, a temporary document supports only communication, not recall or elaboration of the idea, or sharing it with anyone outside the room (not to mention a maintainer, who perhaps five years from now will need to understand the software).

The attitude towards documentation is in part a reaction to many development processes and methods that produce a very large number of documents, requiring plenty of effort to write and to maintain.

Gradually, it has been understood that documentation is needed, but that it is more important to product a system than its documentation.

Now, it is more widely understood that there are more needs for documentation than the team's everyday development process. Some documents are needed to share designs between people, documentation is needed in maintenance, understanding of the architecture requires documents, and there are mandatory requirements to document in safety and quality standards.

In safety-critical development documentation has an important role. But even there, documentation needs to be kept reasonably short. In fact, the standard IEC 61508-1, 2nd ed. states that: "The documentation shall: be accurate and concise; be easy to understand by those persons having to make use of it; suit the purpose for which it is intended; be accessible and maintainable".

In agile development it means that we need to choose such documentation tools that meet these kinds of criteria:

- Support for the volume of total documentation in a usable way.
- Restructuring of information, when the product changes, is easy and fast.
- Writing and drawing of information is easy.
- The tools should support teamwork. For example, how easy it is to collaborate in drafting a design and transfer it into permanent storage.
- Updating of information is easy.
- Configuration management should be as automatic as possible, for example to see what information has been invalidated with product changes and needs to be reassessed and updated.
- Documentation structures are rich, enabling the inclusion of safety information in every place where it is required.
- Documentation needs to be available to all parties for viewing, but changing it needs to be controlled, especially the changing of safety related information.

The basic idea is that resources should be used for communication, analysis and synthesis, and not in mechanical document production. Wikis (Bernier et al, 2010) have gained wide use in software development in Finland due to their flexibility and ease of changing information rapidly. Patterns for their usage in safety-critical developments still need development. Yet, safety-critical development benefits from more complete information systems, that feature linking of requirements to design, implementation and testing; supporting traceability, defined baselines, reviews, acceptance and external auditing,. Such systems are often called application lifecycle management systems. Polarion ALM (http://www.polarion.com/) is one example of these. Such systems are usually designed for traditional design processes, however, and their application in more agile processes will need more analysis. But modern such systems are highly tailorable, which should help in their utilisation.

8.3.1 Process attitude: Programming emphasised over models

One important aspect of lean culture is placing emphasis on coding instead of modelling and designing. There are many viewpoints to that phenomenon.

Agile was a response to previous "heavy" methodologies like RUP, with which developers sometimes got the feeling that they never got to actually implementing the system – they seemed to be doing mostly modelling and other design tasks. Agile emphasises getting working software, and coding is the thing that matters most – without it there will be no running system.

Thus, when the other aspects of development are minimised, the development could be more rapid and the workflow as flexible as possible with the fewest number of tasks and "moving parts". Keeping things simple is an important agile principle.

In fact, models have mostly been used as documents, as model-based development was rarely sufficient enough to create all the code – just a template, and maintaining both the models and the code feels like redundant work.

Code is also something that all developers understand, which makes it possible to practice pair programming and to have shared ownership of code.

And when the agile processes implement things incrementally, small parts at a time, the abstraction provided by models is not necessary to support design – small parts can be understood from their behaviour better than their descriptions. And the stakeholders who participate in the development should not need to understand formal descriptions; they benefit more from working programs or prototypes to be able to participate in the process. They might feel left out if they were required to understand diagrams.

All in all, the emphasis on coding seems to have good reasons. It certainly is not just immaturity or non-professionalism or regression back to the tools of previous decades.

How does safety-critical development relate to this?

In safety-critical development the object of development is not just a shared experience of running the application. Just as much of the development concentrates on the invisible: the readiness of the system to handle whatever disturbances there might be during its operation. Program code does not provide a sufficiently abstraction level for understanding that capability or for discussing and analysing it. Expressions of a higher abstraction level are needed. Various kinds of models and diagrams are the necessary tools for that purpose.

Failure analysis and other functional safety assessment require abstractions that describe the system's behaviour in an appropriate way, and diagrams (such as sequence diagrams or state transition diagrams) are designed to do just that. Models are efficient in displaying problems in design. They support visual inspection where conflicts can really be seen – which is not the case when inspecting source code.

Version 2.0 of IEC 61508 recommends formal methods and semi-formal methods especially on higher SIL levels and that alone means that many kinds of diagrams should be used in development and thus the path to code is slightly longer than in traditional applications of agile development.

This is something that needs to be accepted, but utilised carefully.

Shared objects, by which things can be discussed, learned and assessed, are a key element in agile development. Therefore the following needs to be considered:

- When planning presentation of designs, implementations and system behaviour, we need to first identify the people who need to understand the presentation – be it a text, a graph, a demo, or whatever.
- To support collaboration, if diagrams are used in reviews and discussions, their understandability should be high. Everyone who participates should get a grasp of what the main messages are.
- A running system should be provided by any means practicable. If not the actual system, then a prototype. If not a prototype, then a visual simulation of the system.
- While there is a tendency to keep things lean, there may be redundancy in presentations so that a suitable presentation can be used in each situation. For example, it is widely accepted that architecture requires many views of it in order to have its main features expressed sufficiently. The same principle can be utilised in other development situations.

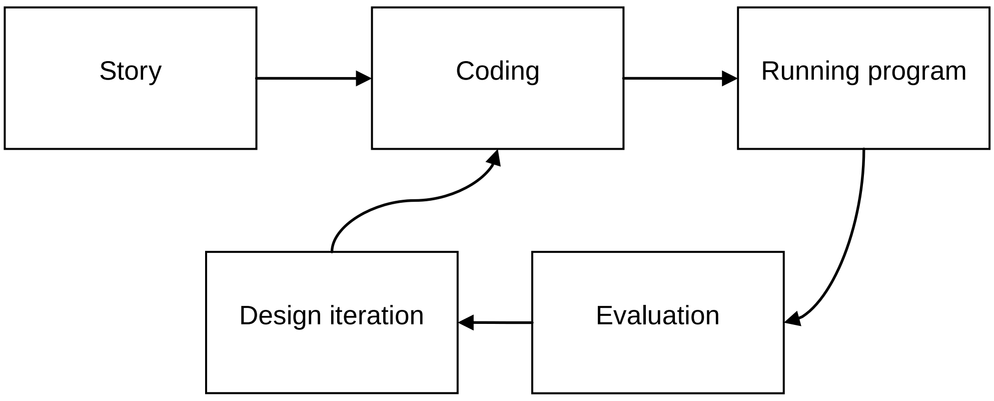

Figure 12. Coding is central to generic agile development.

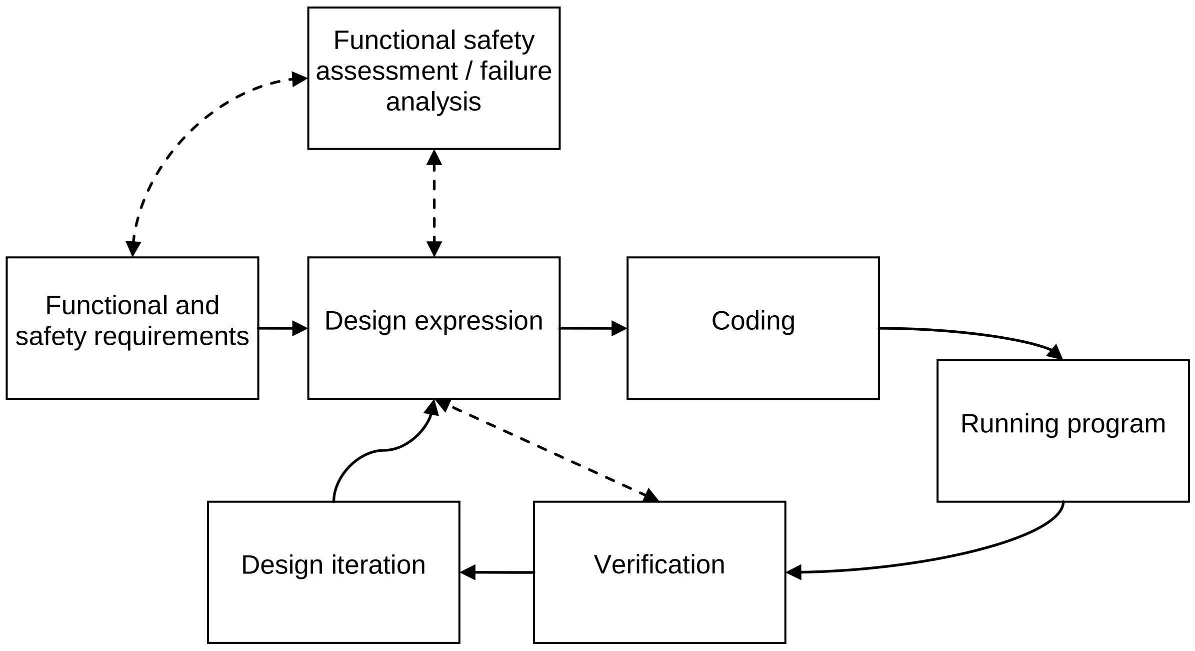
Functional safety assessment / failure analysis
Functional and safety requirements
Design expression
Coding
Running program
Design iteration
Verification

Figure 13. Safety-critical development needs rich expressions.

8.3.2 Organisational feature: Pair programming

Pair programming is a traditional agile working mode where two programmers work on the same code at the same time, at the same workstation – one at the keyboard and the other just discussing, commenting, giving ideas. The technique is quite controversial, if not no other reason than costs – two pairs of hands at one keyboard cannot be that productive. And in fact, studies have shown varying results regarding productivity (slightly higher development cost at 15 % according to Mustafa et al, 2005, but safety-critical development may have special factors that have not been researched sufficiently). But there are many other possible benefits and factors that contribute to the success of pair programming. Mustafa et al (2005) report the following issues: quality, team building and pair management, pair personality, threatening environment, project management, design and problem-solving, programmer resistance, communication, knowledge sharing, mentoring, environment requirements, effective pairs, shared responsibility, human resource management, attitude, morale, productivity, development costs, and enjoyment of work.

In the case of safety-critical development, some of these warrant a closer look.

- Team building and pair management. Safety-critical development can be more engaging teamwork than other kinds of development, as the developers participate not only in development, but also failure analysis and other tasks. So if pair programming helps the team members work together better, it will have a positive influence on safety-critical development tasks also.

- Knowledge sharing. Experienced developers utilise a huge personal databank of information related not only to programming, but hardware, systems usage, known problems, hazards, failure modes, etc. Most of this information has not been documented in any way, but must be transferred to new developers during the course of projects.
- Mentoring. The culture of safety-critical development can be strange to new developers. Pair programming provides a good environment for mentoring, if experienced developers are paired with new developers.
- Shared responsibility. Safety can be a heavy responsibility, but when there is someone else sharing the design decisions (although reviewed later by a team), there should be less pressure and thus better designs and implementations.

So, while pair programming is not generally so much fashion nowadays, it might provide important benefits in many situations in safety-critical development, especially in maintaining the culture of companies and helping new developer generations step into the organisation. Still, productivity is a very practical issue for many development organisations and in that regard, the results may vary, due to many factors.

Sometimes pair programming is thought to be a means to replace code reviews. An important element of any review is independence of the reviewers regarding the code, and as both programmers in the pair are authors, that requirement is not fulfilled.

For more information about continuous integration, see Wikipedia article Continuous integration: http://en.wikipedia.org/wiki/Continuous_integration

8.3.3 Software engineering practice: Refactoring

Code and architecture have always been modified when needed, when, for example, functionality and architecture changes. In the agile world this has been taken further. It is now called “refactoring” and is supported by various refactoring patterns. Mostly, we discuss refactoring here done with the goal of improving of code while keeping its external behaviour the same. Agile development often starts with a first version of code for some function that is to be improved – perhaps to make it more maintainable, more robust or more secure or to remove duplicated code. Of course, code reviews may point out coding patterns that should not be used, or external libraries change, requiring adjustments in the code.

Two important principles help refactoring succeed:

- It is considered a normal, everyday activity, which should be done and done properly.
- (Unit) tests work as a safety net for the modification by ensuring that the behaviour is not changed when the structure changes.

However, it is now understood that refactoring is usually only possible on a small scale, inside a class, a unit etc. Larger level architectural changes are as difficult as before, and even more so, if the overall architecture has not been planned properly.

This situation causes a serious issue for safety architecture. It requires proper planning to be solid and to support later changing of details and refactoring of its implementation. If left solely for individuals to refactor, it may lose simplicity and maintainability – and thus, safety.

Refactoring causes a problem in safety-critical development, because it invalidates at least part of the system and will cause a required revalidation even if the behaviour of the system has not changed. This is sure to be something that an organisation will consider wasteful.

For more information about refactoring, see Wikipedia article Code refactoring at http://en.wikipedia.org/wiki/Code_refactoring

8.3.1 Process issue: Code reviews required in safety-critical development

Another issue related to refactoring is code reviews. Code reviews are required or highly recommended for safety-critical systems. And in safety-critical settings they should be done in a proper, quite strict way: good preparation and sense of responsibility from all participants, as in the Fagan inspection style (see Wikipedia article Fagan Inspection).

But what would be the point of reviews, if the code can and will change soon after the review? No organisation has the resources to review the same implementation over and over. It should be noted that the spirit of safety-critical development is two-fold: 1) assuring the released product, and 2) continuously assuring the process quality. Therefore, one type at one point of time is not sufficient.

Luckily, even standards consider testing as the primary means of verification of code. Code reviews are more recommended for the early design phases and can be used to get implementations started with good quality and using solid engineering principles. In agile development, the early phase means the early phase of each new feature, in whatever project increment. Still, the nature of agile development should mean that code will be changed more than in traditional processes (in agile culture, the term “churn” is used to emphasise this) and thus requires more attention.

There can be various strategies for handling this issue. A combination of the following can be used:

- Manual reviews are used as an exception, not as a rule. Instead, automatic code analysis is used, automatically in the build process. Automatic tools should be used anyway, and some of them can check the code from very many viewpoints.
- Manual reviews are used mainly for new developers in the team and for most critical code and when the programming of new features is started.
- Automatic start of the review process in configuration management, when new or altered code is checked in. A requirement for at least one person to review the code.
- Mandatory pair review could be used in some cases.
- A not so fruitful idea is making a refactoring strategy that will cause changes only at suitable intervals and in a volume the team can handle. This restriction to refactoring during development could bring “natural refactoring” to a halt.

Off-the-shelf automatic code checking tools can be used to assess coding style and to detect many potential problems (such as traditional memory buffer related problems in C programming language or use of problematic language constructs or dangerous runtime / API functions). Furthermore, custom tools can be created to monitor integrity architecture, so that, for example, applications use only the layers they are allowed to. Because of this, manual review should therefore concentrate on things that are not easy to check automatically. These include:

- Generic expert view on potential problems in the code.
- Checking that defensive programming is practiced – for example, all arguments to functions are checked for validity before using them; all return values of function calls are checked and/or exception handling is flawless.
- Checking of the commenting style and naming conventions so that there will be no misconception of the purpose of code.
- Checking that the code is easy to understand and maintain and uses such design that refactoring is easy, should the need arise. Clarity of design pattern and length of methods are two typical issues to check.
- Checking that the coding patterns are as testable as possible so that unit and integration tests are easy to create and maintain.
- Checking that the code is modular, again to support testability and problem-free refactoring and maintenance.
- Checking that the code does not bypass any checking functions, such as turning off compiler warnings or warnings generated by an automatic checker, unless there is a specific allowance for that.
- For new developers in the company, a checking that all rules and company practices are applied properly.

So, manual review of code need not include desktop simulation of programs and methods, but should concentrate on critical issues – a review is not simulation, but more like identification. Creation of a checklist will be of help, as is keeping track of common problems found in reviews – unless they can be prevented or detected automatically, they are likely to appear again.

Still, shared code ownership is one important agile principle, and it enables colleagues to monitor other team members’ code quality, which helps enforce coding standards (highly recommended practice for safety-critical code) and reduce refactoring and maintenance risks. Surrounded by complete quality controls, shared code ownership should be an aid in safety-critical development, too.

8.3.2 Software engineering practice: Continuous software integration and deployment

Continuous integration is a technique where, by using a build server, a new build of the software is created when an update of source code is checked in, unless a build is currently underway. This is a very big difference to the “old way” of creating new versions every couple of weeks or even months.

This approach causes a temptation to trust automated tests, emphasise low level tests (mostly unit tests) and to not to freeze the software ever. These ideas will not suit safety-critical development well. For safety-critical development the following principles need to be present:

- Not all builds are equal. The ones that aim to delivery may need different treatment even during low level integration (module integration).
- Low level integration testing is just a beginning! It really just shows that the software may be testable.
- Software needs to be frozen for validation so that it can be determined what it consists of and that all of it will really be tested.

Continuous deployment is a new agile technique, where the software is automatically transferred to target hardware after a successful build (this is also called hardware integration). The deployments in safety-critical development can be of at least five kinds:

- Deployment to a software simulator.
- Deployment to an electronics unit.
- Deployment to a hardware simulator.
- Deployment to an internal test machine.
- Deployment to a customer machine.

At least the first type can utilise continuous deployment, but the others will require further testing and other verification before deployment.

8.3.3 Process feature: Testing in agile projects

Agile approach to testing corresponds with many traditional elements of good testing:

- Testing is started early on, immediately when implementation begins.
- Unit testing is done systematically using automated tools, making it a low level regression safety net.
- Automation is seen a help and developers implement the automation as a part of their everyday work.
- Low level integration keeps the system under development in good running shape continuously.
- Testers are included in the team, which improves communication and collaboration.

Still, there are problems from the viewpoint of safety:

- Safety is a system level issue that unit level testing cannot handle properly.
- Low level integration testing cannot study systems integration issues.
- The process relies on unit testing being done, but in practice it is not carried out as much as it should be.
- Testing of safety really requires independence from the testers.
- Assuring safety requires many viewpoints – analysis and testing, technical and operational perspective – but agile culture is too homogeneous in that regard.

Therefore, the agile practices need to be supplemented: while everyday testing benefits from testers being included in the team, the releases that are aimed at the customer (even for testing) need to be tested by (more) independent testers. As this can be work that should not be done at the end of every increment, release validation testing can become a layer above the agile increment process.

Agile projects sometimes do not use any kinds of test plans, because it is felt that nothing should be planned because it is not known what will be designed and implemented. That is not true. In every project there are many known factors: the overall concept, basic selection of technology, and idea of division of work between machinery, software and people. And in safety-critical it is critical to do this kind of concept-level designing so that hazards can be assessed. This will mean that a traditional “Master Test Plan” is a very essential part of planning even an agile project, as it outlines our approach of how to do the testing during the project.

Agile development often includes agile system testing, where test specifications are not pre-planned, but are based on new software implementations. The traditional specification-based approach is mostly like this (this is a simplified description):

- Early in the development, before detailed design and implementation, process comprehensive requirement specifications and functional specification are written.
- Based on these, a test specification and a large number of test cases are designed by a test analyst or test engineer.
- When an implementation of the software can be executed, the tests are run by a test engineer.
- Errors are reported in an error database.
- Product quality metrics are updated.

The agile approach does things differently:

- During an increment, new features are selected to be developed.
- The developer and a test engineer discuss testing.
- During appropriate phases the test engineer performs tests based on the implementation.
- Testing can be exploratory (see Wikipedia article Exploratory Testing for more details), based on observations of how the system behaves, or more analytical, based on traditional test analysis.

- Often, tests are simultaneously automated so that they can be run during automated build processes as regression tests.
- Errors are not reported in an error database, but discussed with the developer and corrected as soon as possible.

A problem with the agile approach is that it might leave out documentation or proper logs, but these can be arranged easily, as all necessary test actions should be carried out – they are just executed differently: different impulse, different timing and a different basis for test design.

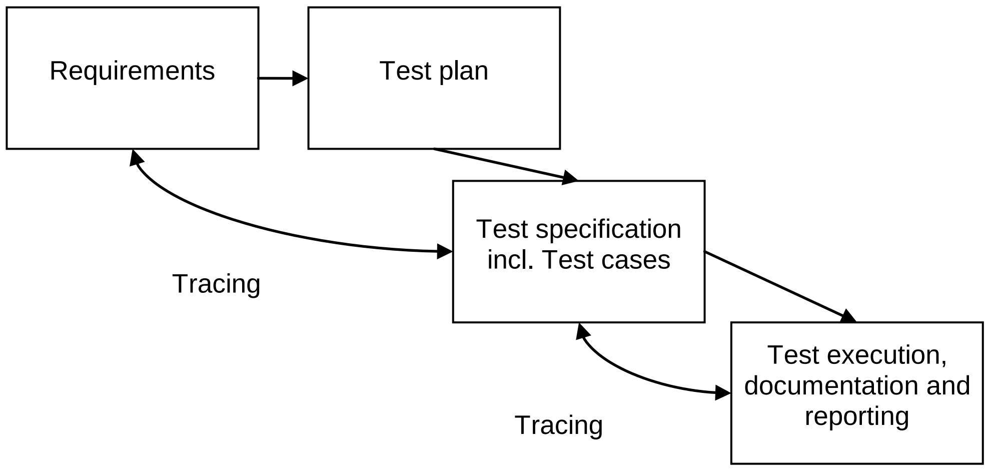
Requirements
Test plan
Test specification incl. Test cases
Tracing
Test execution, documentation and reporting
Tracing

Figure 14. Plan-based testing fully supports tracing.

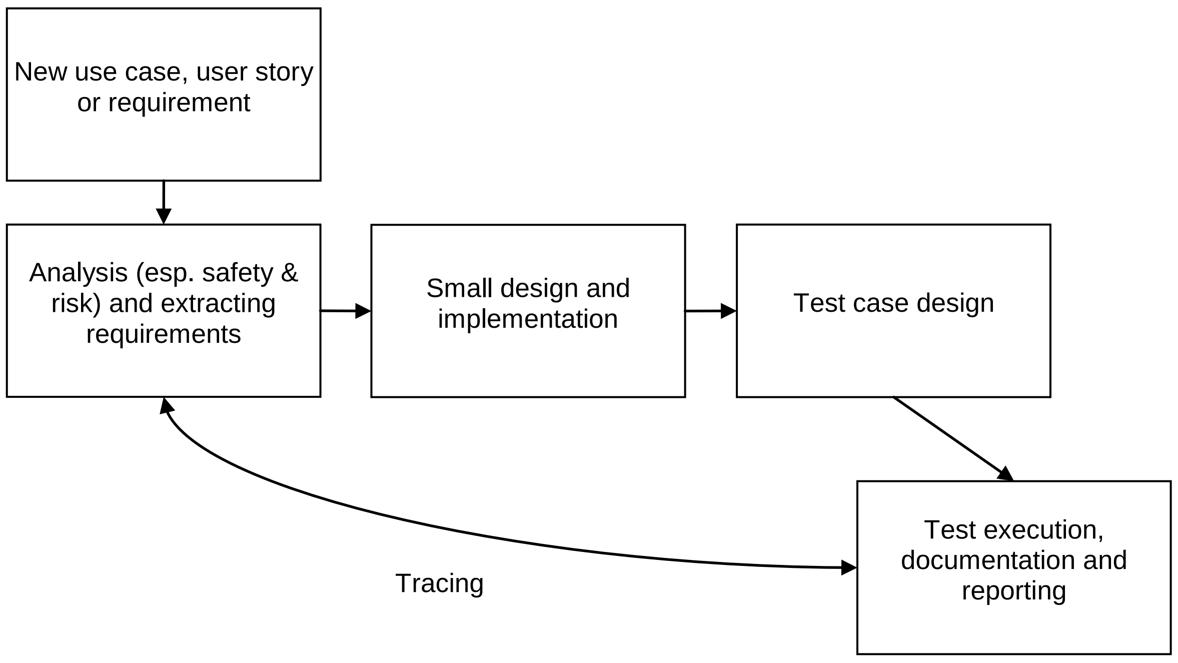
New use case, user story or requirement
Analysis (esp. safety & risk) and extracting requirements
Small design and implementation
Test case design
Test execution, documentation and reporting
Tracing

Figure 15. Agile testing will support tracing if a requirement management system is implemented – that is, there is more to link to than a user story.

Because many safety-critical systems are logic bases, systematic test design, based on the logic, is very natural. In fact, the second edition of IEC 61508-3, 2nd ed. highly recommends that model-based testing is used at module design and integration testing (standard table A.5) – it is recommended for SIL levels 1 and 2 and highly recommended for SIL levels 3 and 4 – and that modelling is used in validating the system's safety (standard table A.7). There is a tendency to recommend more analytical test and validation methods, which makes practical sense, as modern systems are so complex that the tester's thinking and test generation will need help of models and model-supporting or model-based tools.

Nevertheless, elements of agile testing can be included in the process. This also supports the tradition of safety assessments, where one viewpoint only indentifies a certain amount of problems, but new viewpoints are needed to make the assessment more complete. This author has often in testing training proposed the following mix of agile and systematic testing:

- Testing of a new version can start with exploratory testing (see Exploratory Testing, Wikipedia article), where the goal is to make observations of the system and to learn how it behaves. This will help the tester (and developers) to understand the system and to immediately see problems in the functionality. It will also help formulate systematic test cases. This phase does not need test cases, but a planned approach and some use cases to guide the testing. Exploratory testing is traditionally done via a GUI – perhaps a rapid mock-up over a system interface. Modern test tools can make this quite efficient.
- The second phase is the detailed design and execution of planned test cases, developed and selected based on the specification and models, findings from the first phase and the actual implementation. This is the phase of testing that will produce the required tracing of testing to the requirements and also the main assessments of test coverage (again, against the requirements).

- After the systematic test cases no longer find new errors, agile testing is used again, to cover areas that still need more “probing”.

Both agile phases are based on the tester’s skills, experience-based “hunches” and also experiences of where problems could be found in systems of this kind.

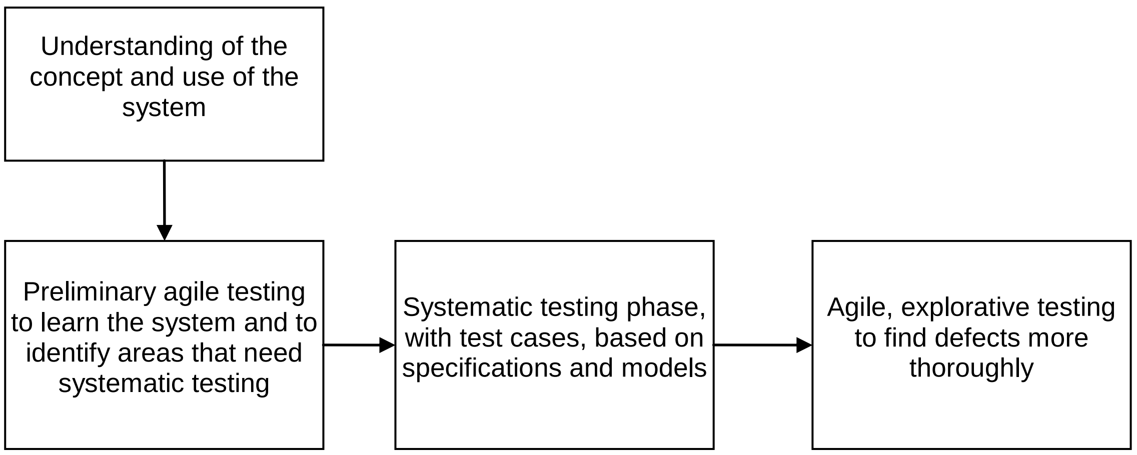
Understanding of the concept and use of the system
Preliminary agile testing to learn the system and to identify areas that need systematic testing
Systematic testing phase, with test cases, based on specifications and models
Agile, explorative testing to find defects more thoroughly

Figure 16. Agile and systematic testing forming a controlled flow of test thinking and execution.

Of course, during all phases, test automation is utilised and developed further, to aid in testing of next releases and in regression testing.

Error management is often quite weak in agile projects, because it is replaced with close collaboration between testers and developers and a low-level approach to quality. That is not sufficient in safety-critical development. Traditional error management information systems are needed, even though they are combined with close collaboration inside the development team.

8.3.4 Process practice: Test-driven development

Test-driven development means a practice, where tests are developed for a feature or a class or other development task before its detailed design and implementation. Tests are developed incrementally, in small pieces – in contrast to the traditional way of writing a full set of test for a specification at one time.

This has good consequences in that tests will remain in place during implementation and they will remain as a safety net for refactoring. The development of a feature will not be considered “ready” until all the tests pass.

This principle can be applied in many levels:

- In module development, where the tests are unit tests.
- In full feature tests, where the tests are system tests (which actually are often called “acceptance tests” in agile culture).
- In systems integration, where the tests are systems integration tests.

Now, in the general agile culture, tests are almost a synonym for verification and validation, so the principle could actually be called "verification and validation -driven tests". Thus, in safety-critical development, the development of a feature should be based on validation so that it meets the requirements and fulfils its function. Depending of the development task, the meaning of this can vary:

- When we are developing a functional feature, the resulting validation consists of general software engineering validation tasks, such as testing and review, but also an updated hazard and risk analysis and safety assessment, so we can assess what the safety level of the resulting implementation is and what safety controls it might need (or an update of understanding this).
- When we are developing a safety system, the resulting validation tests that the control fulfils its requirements, does what it is intended to do and does not do anything else.

So, this formalised principle supports safety-critical development appropriately.

Note, however that while test-driven development at unit level has been seen to be almost a requirement, similar benefits may be produced with other means. Siniaalto and Abrahamsson note that "[…] the present authors query whether the reported external quality benefits can be achieved with a more traditional approach to unit-level testing or whether they are really due to TDD itself."

For more information about test-driven development, see Wikipedia article Test-driven development http://en.wikipedia.org/wiki/Test-driven_development

8.3.5 Organisational feature: Who does test automation?

In agile development there is a tendency to do all the development inside the teams. But in distributed projects it may result in wasted resources and incompatible systems. Separate test automation teams may be one solution (as noted by Woodward et. al., 2010). A separate test automation developer team can support many development teams and has a concurrent view of the total system, at all abstraction levels. It enables the software developers to concentrate on what they should really be doing: developing good, safe software and not test infrastructure.

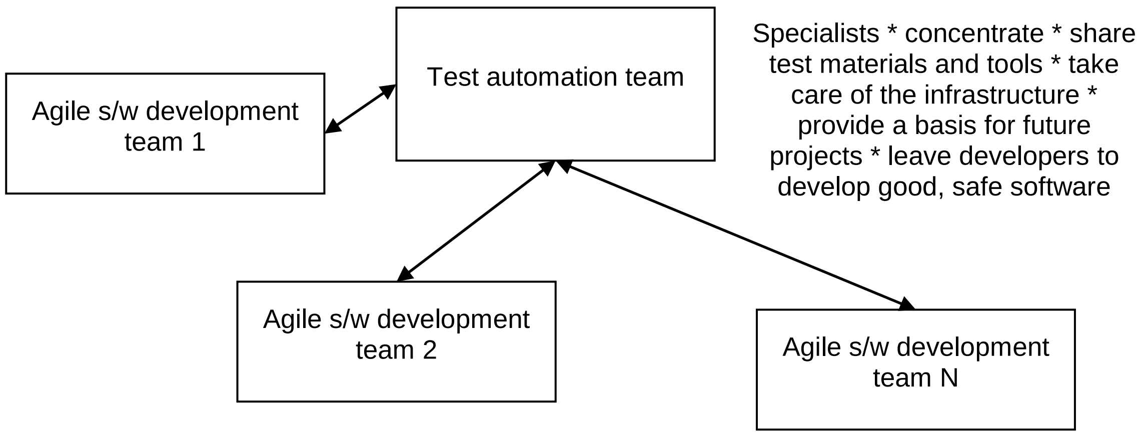
Test automation team
Specialists * concentrate * share test materials and tools * take care of the infrastructure * provide a basis for future projects * leave developers to develop good, safe software
Agile s/w development team 1
Agile s/w development team 2
Agile s/w development team N

Figure 17. A test automation team serves all software development teams.

This issue provides an important lesson: in a distributed project not all teams need to be equal, but can specialise in certain issues and in that way benefit the whole company, especially in the long term.

8.3.6 Quality goal: Keeping the defect count in zero during development

Together with continuous integration, the projects often employ a goal of keeping the defect count during development at zero – or at least as low as possible. Of course, this can only apply to defects that can be measured, so in practice it may mean those defects than can be found in automatic tests executed in the integration process. Even that will mean that each build will produce a software version that can be executed and that can perform most tasks as expected. Often, the results can be masked because higher-level defects are not reported – testers communicate those directly to the developers. And of course, this principle requires good testing.

The research on safety cultures shows that if we aim at zero accidents and incidents, there is a growing psychological pressure not to report any findings that might challenge the goal. Therefore, in order to succeed, this principle needs support from the company culture where each new defect finding is received positively. A defect, a failed test, means that we may have met a new phenomenon which we did not know before – and now we can improve!

But still, as a technical goal, this is one practice that will make development of safety-critical software easier, because the stable software base and incremental defect correction will mean that there will befewer surprises when higher level tests show a need to make correction.

Stability is so important in agile development that if there are problems in the project, a whole increment can be used for getting the project back into a stable state. Usually, this results from a too large number of uncorrected defects or a need to change the architecture, but in safety-critical development, a reason could also be, for example, a revision of the safety architecture before continuing or doing any safety and reliability analyses that were for some reason not done already. This “correction increment” practice is more suitable for processes that use reasonably short increments – during a 30 day increment, development should be able to proceed again.

8.3.7 Process attitude: Positive attitude to automation

Agile culture has a remarkably positive attitude to all kinds of automation, including test automation, integration automation, automated deployment, progress dashboards etc. The main reason for this is that under rapid increments, development needs help from automation to ensure that everything is in order. One good example of this is UI-level testing, which has always been difficult to automate and which has been the testers’ responsibility. In agile development, the developers welcome any challenge to automate UI tests too. All this is very valuable for safety-critical development, because we need to let machines do the repetitive things and have the best people concentrate on things that require thinking, such as safety and risk analysis, failure analysis, change management, keeping the architecture in shape, etc.

Another very important reason is the time needed for verification and validation. Testing needs to be very thorough in safety-critical development and if its automation level is low, it can be quite time-consuming, perhaps causing a bottle-neck in the validation and certification process.

Agile teams use automation in simple ways, utilising tools that are simple to use. This is in stark contrast to traditional engineering systems, where big systems consisting of large application are used to control the process and manage requirements, testing, faults, etc. But while systems should be simple, they should not be too simple. Whereas many non-safety-critical agile teams can manage many tasks using Excel sheets, etc. that is not realistic in safety-critical development due to the many issues that need to be managed and linked together.

9 Agile and Lean

9.1 Origins of Lean

Lean is an approach that is often associated with the context of agile development. Lean was originally called “Lean production”, but today “Lean” represents only a vision of good efficient ways of action. It has been applied mostly in manufacturing industries, but lately it has been coming into the software development world (see, for example, Poppendieck 2007). Lean originates from Toyota’s car manufacturing, where some new principles were identified that enabled Toyota to be fast and flexible in its car production (a rapid changing of the model to be manufactured or change in the configuration of the car). It was also inexpensive (low capital and quality costs), efficient (fault-free, optimised production and logistics, no learning curve in production) and would produce faultless products.

In its purest form, Lean is documented as ”The Toyota Way” (The Toyota Way, Wikipedia article) and its most concise expression are its 14 Principles. Those are included in Appendix 1. The principles are elaborated more in, for example, a book by the same name (Liker, 2004), available in many languages, including Finnish.

9.2 Lean in software development and compared to Agile

Mary and Tom Poppendiecks have been the most active in transforming the principles into the software development world. They have formed the following guiding principles (condensed from Poppendieck 2007):

- **Eliminate waste**. Understand what your customers value. Create nothing but value. Write less code.
- **Create knowledge**. Create design-build teams. Maintain a culture of constant improvement. Teach problem-solving methods.
- **Build quality In**. Synchronise tasks from the start. Automate so that task become routine. Do it in a way people can improve the process. Make it possible to change anything. Refactor and eliminate code duplication to zero.
- **Defer commitment**. Schedule irreversible decisions at the last responsible moment. Break dependencies between components. Develop alternative solutions.
- **Optimise the whole**. Focus on the entire value stream and the whole organisation, not on single processes. Deliver a complete product.
- **Deliver fast**. Work in small batches and short release cycles. Limit work to capacity. Put in your task queue small tasks that cannot clog the process for a long time.
- **Respect people**. Train team leaders/supervisors. Move responsibility and decision-making to the lowest possible level. Foster pride.

The principles are more agile than the original Toyota’s principles and can be used to develop agile practices inside an agile project model. But still, Lean is not the same as agile. They have very deep differences, which is condensed in Coplien (2010) into the list in Table 4.

Table 4. Contrast between Lean and Agile (Coplien, 2010).

Lean	Agile
Thinking and doing	Doing
Inspect-plan-do	Do-inspect plan
Feed-forward and feedback (design for change and respond to change)	Feedback (react to change)
High throughput	Low latency
Planning and responding	Reacting
Focus on process	Focus on people
Teams (working as unit)	Individuals (and interactions)
Complicated systems	Complex systems
Embrace standards	Inspect and adapt
Rework in design adds value, in making it is waste	Minimise up-front work of any kind and rework code to get quality
Bring decisions forward (Decision Structure Matrices) (*	Defer decisions (to the last responsible moment)

*) Note: in his presentation at Tampere University of Technology, in 2009 (Coplien, 2009), Coplien explained Lean's approach to the timing of decisions as: "Letting a decision go beyond the point where it affects other decisions causes rework, so bring decisions forward to a point where their results don't propagate".

The first item on the table captures a lot of the spirit of differences between agile and Lean: Lean is built on thinking, whereas the core of agile is in doing. As the "thinking" part has been seen to be lacking in many agile development projects and practices, Lean can bring the missing parts into the process.

In practice, Lean is sometimes understood as a mechanistic "waste hunt", in which documents and planning are discarded as artefacts and work that does not produce value, but nothing could be further from the truth. Lean encourages planning to make things more efficient in execution. Lean understands that documents are necessary to capture information, but the main use for them is to share knowledge between all people in the company. A team might need fewer documents, but in Lean we think of the whole, and in the long term.

But when we are transferring the solid safety-critical development practices into a more agile way, the thinking processes are already in place, and there should be caution in bringing more development paradigms into the equation. Indeed, both Lean and Agile respect keeping things simple – "as simple as possible, but not any simpler".

Of the many principles of Lean, the following should be especially noted in the safety-critical context (these are freely adapted from the principles):

- Leadership and management support for safety.
- All-encompassing safety and quality culture.
- Respect for expert knowledge and skills on safety.
- Use of analysis and analytical tools.
- Aiming at efficient standardised processes.

- Continuous process improvement (on top of the solid standard processes).
- Flexible tools that can respond to change and can be taken fast into use in a project or its phase (a new increment that needs adjustments to the tools' configuration or other preparation).
- Full understanding of what customers need – full understanding of product’s use and its expected safety features.
- Welcoming defects in the development phase – they were not left for the users to find.
- Deep problem analysis and changing of activity to prevent problems in the future.
- Helping of subcontractors reach the same level and same thinking as us.
- Continuous long-term development of all areas of activities (Kaizen), not just development or verification process improvements.

All in all, Lean is not a process, but a way of thinking, shared by every member of the company and its subcontractors.

9.3 Analysis of some important Lean practices

It should be noted that implementing Lean fully requires a holistic approach. It does not contain a “silver bullet” of company transformation. There are many instances where companies have failed in Lean when they have misunderstood the approach and have tried to implement Lean in a mechanistic way, leaving out important ingredients.

But still, Lean has brought into agile culture some important practices that can be implemented individually, to produce benefits. We will look into the most important ones shortly.

9.3.1 Visual process monitoring and Kanban

Lean aims to support immediate seeing of the status of a process. In the original Lean manufacturing this principle of “Kanban” was used in production workstations that had visible flags telling the status of that post – whether there were problems or if it could use more parts or materials. This is one form of the “pull” principle – responding to a display of need before action.

In development work, this visual system is exemplified with visual tracking of tasks in a shared display –a task board on a project room wall with “yellow notes” representing things under development (requirements, functions, use cases etc…) that are moved from phase to phase (for example to do -> design -> implementation -> testing -> ready).

Not started	In development	In testing	Done
			Architecture
		Comms	
	UI		
		Reports	
		Transactions	
Manual			
Installer			

Figure 18. A wall board showing progress of tasks in an agile team's room (a simplified visualisation).

Such functions can, of course, be implemented with information systems, especially on distributed projects. Another means of visual communication is the use of dashboards for giving shared concise information of the project's progress – especially to management. In order to the project to be efficient, this information needs to be synthesised automatically from project tracking systems.

☺	☺	☹	☹	😐
Functionality	Safety	Usability	Documentation	Overall

Figure 19. The concept of a dashboard (a simplified visualisation).

Of course, such real-time information does not replace reviews, but it should make critical status information clear to everyone. This is a principle that can be used in any kind of development process, not just agile.

Obviously, the more safety-critical a product or system is, the more clearly we need to display the state of its safety design so that we know when it is ready for validation or release process.

9.3.2 Short, readable reports

As Lean aims at sharing information between all parties effectively, it supports short and concise reports that can fit on a large printout or one computer display. This is a principle that is valuable for communicating with management – “management summary”, but obviously, technical reports, specifications or risk analysis reports contain valuable information that simply requires a certain number of documentation pages.

For any safety related activity it is essential that the information provided is read and understood, so, when processes are changed, the quality of any reports used should be carefully assessed.

9.3.3 Decisions on the production “floor”

Because production needs to be efficient, problems must not be passed forward to the next project phase, but instead, the process can be halted and problems solved before continuing. In a lean factory, everyone can stop production when such a need arises.

Of course, as production is done in teams, not everything needs to be stopped – that would be stupid – but just a flow through a certain process.

This is a needed element in modern software development. The team needs to assess the situation every day and react to problems. No matter what the managing principle is, a discussing approach is emphasised in Lean and in Agile, because it is understood that without power and respect, the teams are not efficient.

In practice, the stopping of the process can happen in various ways:

- If integration tests are noticed to not work, the situation needs to be corrected immediately and the developers should help in correcting the problem before continuing development.
- The development team and the test team should have a vote on not releasing a product, even if it meets all the mandatory requirements, if they feel that it is not (yet) fit for use. Agile processes provide good environment for this kind of discussion at the end of all increments, and also a good controlled way of making the necessary changes in the next increment.
- One way of stopping the process flow in agile development is to dedicate an increment to correcting errors and problems and doing maintenance tasks. If the unresolved software defects start piling up, the situation needs to be corrected and an agile process with its increments provides a good environment for that.

9.3.4 Analysis and problem solving in teams

When there are problems, such as a defect found in a very late testing phase or a new kind of defect, it needs to be analysed:

- Why did the defect get this far?
- Could it have been found in earlier testing tasks?
- What was the cause of it? Specification, lack of communication, errors in implementation or what?
- What could be done about it? How can we change our way of acting so that the reoccurences of this kind of defect can be minimised?

This kind of analysis should be done in every kind of development, for example, when problems are found in the validation phase and the correcting of these will need repeating some of the process needing time, resources and costing money. What is special about Lean is that instead of a single expert doing the analysis, the team does it together, which will make process improvements much easier.

Tools are used in the analysis, such as cause-consequence diagrams and this tool/method-based approach should be part of the culture in every development organisation.

For more analysis of Lean's approach to improvement and minimising defects, see Vuori (2010a; in Finnish).

9.3.5 Learning organisation

A lean organisation should be a learning organisation. Agile processes usually contain a self-reflection task at the end an increment, but that is mainly for the team to be able to adjust its behaviour in the next increments and there is no process to transfer the learned things to the rest of the organisation (not that there is anything to prevent it). So, the learning is mostly internal, adaptive and brings small, continuous improvements. That can produce good results if applied properly.

Lean, on the other hand, is based on the idea that the whole organisation should learn from what one team has experienced. And the increment-based agile process provides a good rhythmic basis for that, very different to traditional projects, which publish their lessons learned and reusable components, etc. perhaps only after a project has been ended.

So, it is possible to build around an agile process functions that help teams spread their new technologies, new process improvements, new development and testing tools, etc. to the rest of the organisation.

That was the process part of the equation. Just as important is Lean's understanding that excellent people are the most important asset. The best product needs the best product developers. The best processes need the best people to plan them. And the best collaboration helps everyone get the best out of themselves and others.

The most important reflection of this to safety-critical development is that the possibilities and the rhythm of agile process can be of benefit in spreading safety information, experiences and technology from and to the teams far more effectively than in non-agile processes.

9.3.6 Approach to risk taking in development

Agile, lean and safety-critical cultures differ in their approach to taking risks. By risks we mean issues of:

- Dealing with uncertainty.
- Accepting the failure of a development decision.
- Accepting the insufficiency of a product iteration. This includes safety and security.

The relation to risks of these three cultures is compared in Table 5.

Table 5. Comparison of risk-taking approaches of Agile, Lean and safety-critical cultures.

Issue	Agile	Lean	Safety-critical development
Experimenting	Just try it – it is ok to fail; you can do it again	In product development, new concepts are very welcome	Find something already proven, then assess it and test it
Alternatives	Alternatives are not sought.	Create redundant alternatives	Alternatives are needed for diversity, exhausting use of alternatives at concept level
Design failures	Failing is ok – it helps us learn – just adjust at next increment	Not accepted. Carefully validate designs before implementing.	Failing is absolutely not ok, as someone might get hurt
Phase of risk taking	All development phases	Concept design. Risk taking stops at the design phase.	None
Co-operation and risks	The social process produces compromise – medium innovation risks, medium safety risks	Do the design well	Sense of responsibility is high, making people cautious. Risks are assessed in collaboration.

Clearly, the bias of the agile culture is very much different than that of the safety-critical culture. But this is where the principles of Lean can bring balance to the equation.

9.4 Cultural compatibility

Lean has its origins in Japan, and agile culture and common agile processes have had plenty of influence from Japanese collaboration culture. Western professional culture can be quite different and the processes and practices of safety culture represent western ideals of excellent engineering. Nonaka and Takeuchi (1995) have compared key elements of Japanese and Western professional cultures and their analysis can help us understand the barriers to agile in engineering companies.

Table 6. Comparison of Japanese-style vs. Western-style organisational knowledge creation from (Nonaka and Takeuchi, 1995).

Japanese culture	Western culture
Group based	Individual based
Tacit knowledge oriented	Explicit knowledge-oriented
Strong on socialisation and internalisation	Strong on externalisation and combination
Emphasis on experience	Emphasis on analysis
Dangers of “group think” and “over adaptation to past success”	Danger of “paralysis by analysis”
Ambiguous organisational intention	Clear organisational intention
Group autonomy	Individual autonomy
Creative chaos through overlapping tasks	Creative chaos through individual differences
Frequent fluctuation from top management	Less fluctuation from top management
Redundancy of information	Less redundancy of information
Requisite variety through cross-functional teams	Requisite variety through individual differences

The characteristics associated with Western culture do not necessarily favour agile development. Some observations:

- Individual based and individual autonomy: Agile teamwork favours teams over individuals.
- Explicit knowledge-oriented: Agile is more oriented towards tacit knowledge and creation of it.
- Strong on externalisation and combination:
- Emphasis on analysis; danger of “paralysis by analysis”: As noted by Coplien (2010), agile by nature emphasises doing over analysis.
- Less redundancy of information: according to Nonaka and Takeuchi (1995), redundancy and flow of information are required for the team to learn and innovate together.

This implies that when the processes are developed to be more agile, we need to address cultural issues also. Conflicts need to be identified and practices chosen so that they suit the company culture, and the company culture needs to be changed. But the changing of culture can be hard and that is outside of the scope of this paper.

Table 7 presents more details to the cultural comparison that help us understand the challenges.

Table 7. Comparison of Japanese-style vs. European-style product development of high-end cars (Nonaka and Takeuchi, 1995).

	European-style	Japanese-style
Goal	Pursuit of superior performance	Adaptation to changing needs
Product appeal	Function (e.g., high-speed performance)	Image and quality
Product concept creation	Clear-cut decision at the initial stage, adhered to throughout the ensuing stages	Vague at the initial stage, modified and altered in ensuing stages in accordance with changes in needs
Flow of activities	Sequential approach	Overlapping approach
Ensuing process	Specific design targets fixed at the initial stage are pursued under a strict division of labour	Close cooperation among all departments concerned during the development
Organisation	Organisation according to function and often under a project leader with limited authority	Matrix- or project-team-type organisation under a project leader with authority over the entire process from planning to production to sales
Strengths	Conductive to a relentless pursuit of superior performance, function and high quality	Shorter lead time (3-4 years), high quality, and attuned to the needs in the market
Weaknesses	Longer lead time (7-8 years), high development costs	Risks of compromise on a low level; not conductive to an all-out pursuit of superior performance

The European-style indeed supports the current safety-critical processes very well.

10 Mapping of IEC 61508 required and recommended development tasks to the simplified agile process model

10.1 General

In this section we combine the simplified agile process model and present one way of mapping the tasks and requirements of IEC 61508, 2nd ed. into it. The goal is to show how the volume of safety-critical tasks can be implemented in the work-division structure of agile development and what should be considered in their application.

In the mapping we use the following principles:

- For each process phase we outline its goal, its inputs and outputs, so as to structure the general flow of information as clearly as possible.
- The tasks for the phase are listed in a concise list (a table) in which we aim to show the volume and type of required or recommended tasks.

Notes on markings on the tables:

- Mandatory tasks, practices or techniques are marked with (M)
- Highly recommended tasks, practices or techniques are marked with (HR)
- Recommended tasks, practices or techniques are marked with (R)

In some cases, the terms used have been changed to more understandably reflect the software development culture. Any guidelines presented in the tables are concise simplifications, and thus the reader is instructed to study the original standard text to fully understand what is required during the tasks. For that purpose, the tasks include the sub-clause of an IEC 61508 series standard which will provide a full description of the process requirements.

To place the safety-critical task in context, the tables also contain normal project and process-related tasks, but these are not included completely and only referred to. This is because the actual tasks depend on the concrete project model used.

The scope of the table is safety integrity level SIL-1 unless otherwise noted.

This description has been made for illustrative purposes only and any actual process specification should be done using the original standard documents. The description is made in such a way that its modification and additions in further research or in practical process development should be feasible. Therefore, we hope that companies could use the analysis presented as a solid starting point and support for their own process development.

10.2 Preparation of the process model and the operational system

This is a generic phase in the lifecycle of a company in which the organisation creates preparedness for carrying out projects. It includes many planning tasks:

- Planning of the project model.
- Planning of the software lifecycle(s).
- Creation of information systems.
- Creation of document templates or templates in the information systems.
- Creation of instruction for tasks, especially any safety-critical tasks. The creation of instructions and templates shall carefully consider all relevant safety standards.
- Approval of software development and testing tools and methods that may be used in projects.
- Development of coding guidelines used in projects.

It also includes training of the tasks and methods to the personnel. It should also include sufficient safety training, including safety lifecycle during development, safety requirements, safety design, hazard and risk analyses and safety assessments. Training should include management training.

The preparation phase is especially important in agile development because it does not include long process phases, which could start with the required preparation tasks.

The first table of tasks extracted from of IEC 61508, 2nd ed. contains essential required tasks that need to be carried out outside projects, so that the projects can be started rapidly and efficiently:

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Validation of project's development and other tools (M)	Validation of tools, programming languages, compilers, static analysers, configuration tools etc. that the projects can choose from. Note: some tools that may be developed during the project include specific simulators and other tools that are dependent on the design of the product. Validation results need to be recorded and manuals need to be created for the tools to ensure their proper use. As a special note, "compatibility of tools of an integrated toolset shall be verified" (sub-clause 7.4.4.9). In the agile context this applies besides traditional IDEs, for example, tools used in continuous integration and integration testing in a build server environment.	IEC 61508-3, 2nd ed., sub-clause 7.4.4
Development of coding standards (M)	Creation of unit / product line coding guidelines. When devising the guidelines, tools should be acquired or developed that can be used in automatically checking as much of the coding standard as possible (either in the developers' editors or during the build stage).	IEC 61508-3, 2nd ed., sub-clause 7.4.4.13

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Planning of documentation system (M)	Planning of a flexible documentation system that contains information, for each phase of the development, necessary for effective performance of subsequent phases and verification activities, safety assessment. Note: This applies to the overall system.	IEC 61508-1, 2nd ed., sub-clause 5.2.1; annex A
Specification of responsibilities regarding safety (M)	Specification of the responsibilities in the management of functional safety of those who have responsibility Note: This applies to the overall system.	IEC 61508-1, 2nd ed., sub-clause 6.1.1
Appointing responsibility of safety tasks (M)	Appointing responsibility for safety tasks (coordination, development lifecycle phases, safety assessments etc…) to a person. The software development should probably have a dedicated person in close collaboration with the developers.	IEC 61508-1, 2nd ed., sub-clause 6.2.1
Specification of safety-critical tasks	Especially in agile development, there needs to be a well-thought out toolbox on methods and techniques that are used as needed.	IEC 61508-1, 2nd ed., sub-clause 6.2.5, 6.2.12
Company / unit wide training on safety issues	Assuring that all people who are candidates for projects, have sufficient competence in the safety related duties they will perform	IEC 61508-1, 2nd ed., sub-clause 6.2.13
Keeping track of safety related competences, experiences and training	Include safety-related information in a company-wide CV system. Update the information after every project and training.	IEC 61508-1, 2nd ed., sub-clause 6.2.13

10.3 Software development in the context of overall system development

An important preliminary task is to understand the context of development – the overall system, its development process and what the role of software development has in it. The relation of software to the overall system may vary, but in every development situation it is important to understand how the IEC 61508 standards series sees it. Therefore, we have drafted simplified versions of the diagrams included in 61508-3, 2nd ed., showing in turn:

- The overall system safety lifecycle – safety of the whole system in operation. A machine, device or other, in its planned operation environment.
- The safety lifecycle of electrical/electronic/programmable electronics in the system. These will include (from our viewpoint) most importantly all the programmable logic elements and embedded computers in the system, which will be executing the software that the software development process creates.
- And, in the context of both, the safety lifecycle of software, which is the topic and main interest of this report.

The diagrams should help all participants of software development to understand the role of software: it does not work alone. Software developers need to understand the overall picture in order to be able to develop good, valuable, safe software.

The diagrams are in the original non-agile form. We do not aim to transfer the diagrams themselves into any other form, but will analyse how the various items in the diagrams can be implemented in an agile process.

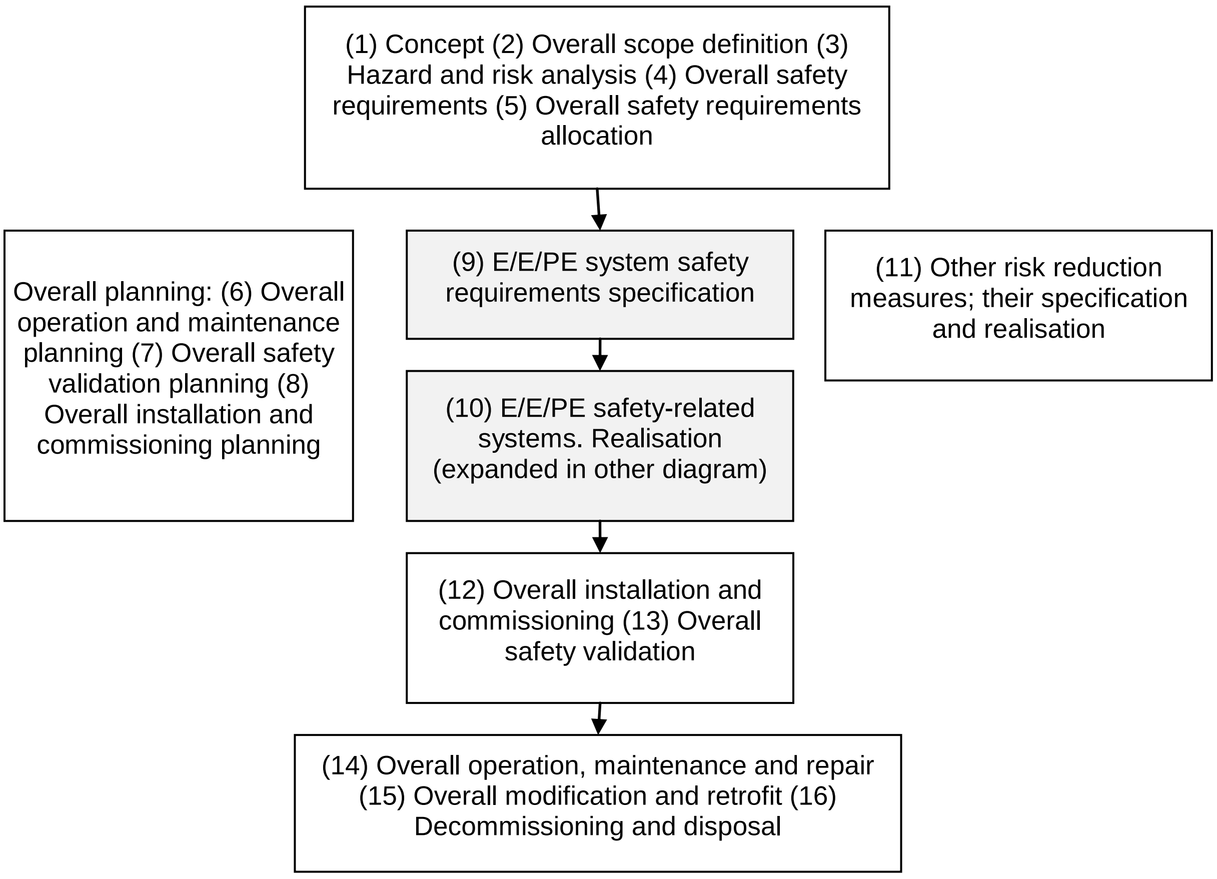
(1) Concept (2) Overall scope definition (3) Hazard and risk analysis (4) Overall safety requirements (5) Overall safety requirements allocation
Overall planning: (6) Overall operation and maintenance planning (7) Overall safety validation planning (8) Overall installation and commissioning planning
(9) E/E/PE system safety requirements specification
(10) E/E/PE safety-related systems. Realisation (expanded in other diagram)
(11) Other risk reduction measures; their specification and realisation
(12) Overall installation and commissioning (13) Overall safety validation
(14) Overall operation, maintenance and repair (15) Overall modification and retrofit (16) Decommissioning and disposal

Figure 20. The overall safety lifecycle. A simplified version of figure 2 of IEC 61508-3, 2nd ed. to show the role of software development.

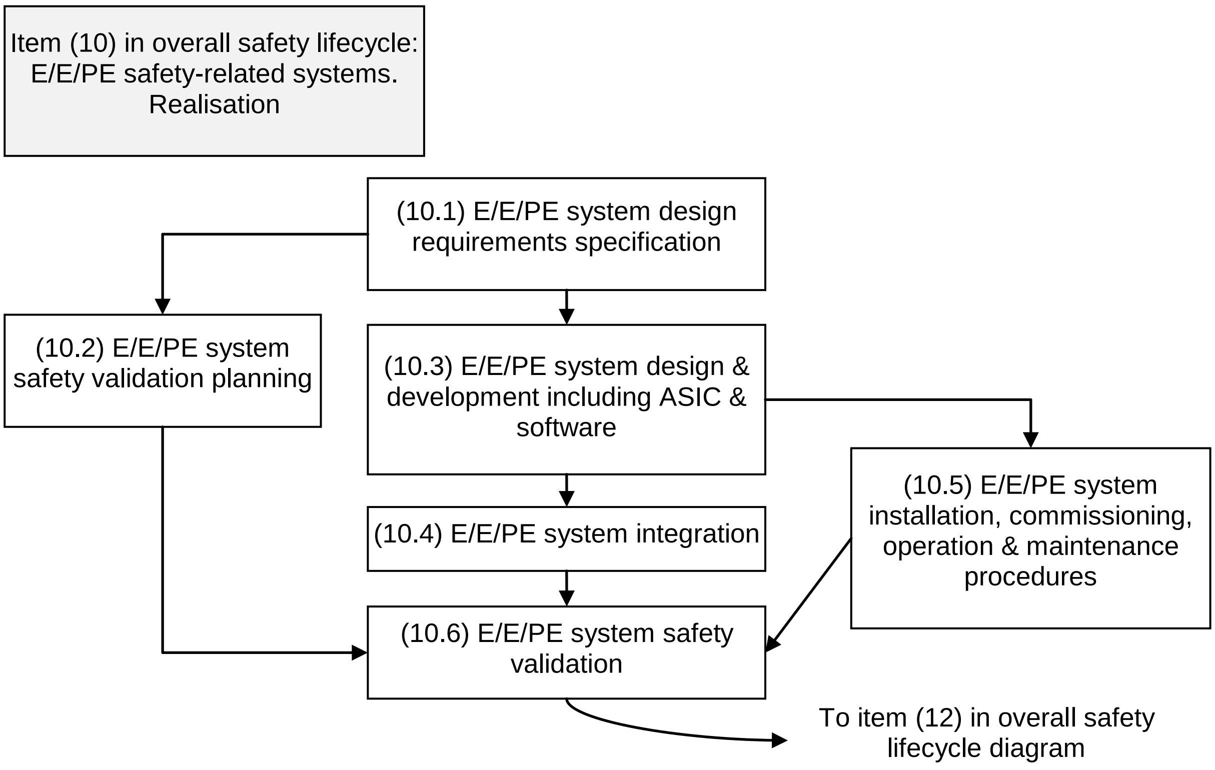
Item (10) in overall safety lifecycle: E/E/PE safety-related systems. Realisation
(10.1) E/E/PE system design requirements specification
(10.2) E/E/PE system safety validation planning
(10.3) E/E/PE system design & development including ASIC & software
(10.4) E/E/PE system integration
(10.5) E/E/PE system installation, commissioning, operation & maintenance procedures
(10.6) E/E/PE system safety validation
To item (12) in overall safety lifecycle diagram

Figure 21. The E/E/PE system safety lifecycle. Expanded box 10 of the overall system safety lifecycle diagram. Redrawn from figure 3 of IEC 61508-3, 2nd ed.

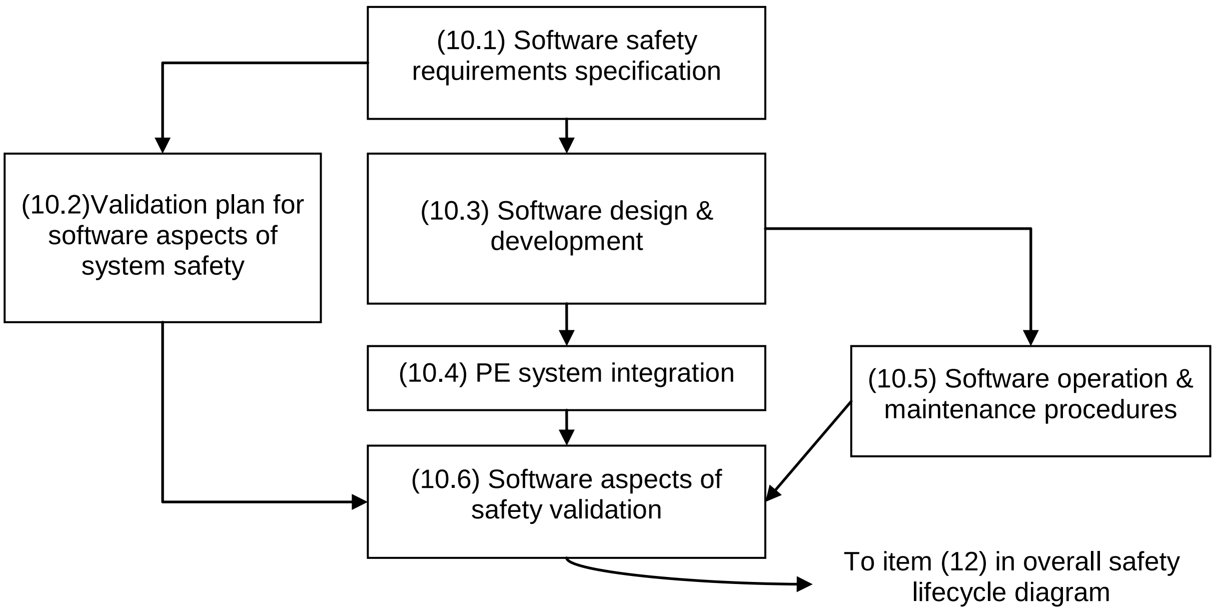
(10.1) Software safety requirements specification
(10.2)Validation plan for software aspects of system safety
(10.3) Software design & development
(10.4) PE system integration
(10.5) Software operation & maintenance procedures
(10.6) Software aspects of safety validation
To item (12) in overall safety lifecycle diagram

Figure 22. The software safety lifecycle. A part of the E/E/PE system safety lifecycle and thus also part of box 10 of the overall system safety lifecycle diagram. Redrawn from figure 4 of IEC 61508-3, 2nd ed.

10.4 Start-up activities

This is the phase in the project that precedes the actual development process. In this phase we need to determine what kind of system we are developing, what kind of a development process and safety processes it requires and then to make decisions based on that. And then start the project.

The activities are focused on the overall system and aim to understand the concept and hazards involved. An important practical detail is the determination of SIL level, which will influence how strict quality and safety management needs to be in the project.

Also, for the agile development lifecycle, we reach an understanding of the nature of agility. Should we have an innovative approach to the project or is the agile process a means of controlling software production or is the main approach release oriented – the customers need early releases for productive production. These choices determine the style of development, but do not influence the standard-based requirements.

Goals:

- Reach an understanding of the development tasks.
- Formulate a description of the product concept.
- Understand the product's risks.
- Formulate the required SIL level for the software system.
- Formulate a project plan. Note: this can be mostly based on a pre-planned process – a project model, an agile software development model.
- Formulate a release plan (a rough level plan of customer releases; most important: how often does the customer wish to receive new versions to production). This may be a total system level decision which is just a constraint to software development.
- Planning of how safety is handled and assured during the project. Note: this can be mostly based on a pre-planned process.
- A solid start to the project.

Inputs:

- Total system level concept.
- Total system level hazard and risk analysis.
- Total system SIL level.

Outputs:

- Preliminary software concept (to be elaborated in the first sprint) in relation to the total system.
- Hazard and risk analysis for the software system.
- Required SIL level for the software system.
- Project plan, safety plan and validation plan.
- Specified owners of product features, stakeholders and responsibilities (especially safety-related ones).

- A project organisation that is able to function.
- Project documentation and information systems in working order.

IEC 61508 tasks

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Making a safety plan	A safety plan and to supplement a project plan will document the safety-critical tasks to be carried out in a project. Note: As agile development will carry out tasks in an incremental and iterative manner, the actual flow of tasks needs to be documented. The structure of this report could present one approach to such documentation.	IEC 61508-1, 2nd ed., sub-clause 6.2.5, 6.2.12, Ch 8 for safety assessments
Planning the safety assessments	Based on the SIL level, it needs to be decided who or what organisation will carry out the functional safety assessments. This is important, because independence is required for validation and the person doing the validation cannot participate in the development process.	IEC 61508-1, 2nd ed., sub-clause 8.2.1, independence: 8.2.18
Selecting the project team (M)	Assuring that all people who will be selected to the project, have sufficient competence in the safety-critical duties they will perform. This needs to be documented.	IEC 61508-1, 2nd ed., sub-clause 6.2.13, 6.2.15
Concept [design] and scope definition (M)	The total system concept, including hardware, operating environment etc. Scope of developed control system and hazard analysis. Note: This is often mostly input to the software development, but in agile development, software developers should be better engaged with this task, in order to better understand the overall system.	Concept: IEC 61508-1, 2nd ed., clause 7.2 Scope: IEC 61508-1, 2nd ed., clause 7.3
Hazard and risk analysis (overall system) (M)	A risk analysis of the overall system and its concepts, using appropriate methods. Note: This is often mostly input to the software development, but in agile development, software developers should be more engaged with this task, in order to more understand the overall system. It should be noted that the risks considered include information security risks, which clearly are of high concern in software development. In agile development the analysis needs to be updated in each development increment, as the system evolves. Document output: Risk analysis report Knowledge output: The generic risk level Other output: A team oriented to safety requirements	IEC 61508-1, 2nd ed., sub-clauses 7.4.2.1, 7.4.2.3

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Determine safety integrity level (SIL) for the system (M)	Determination of SIL level, based on hazards, their consequences and probabilities. On agile development, this may change later when the overall system evolves.	IEC 61508-5, 2nd ed., SFS-EN 62061 A.6
Software safety requirements specification (M)	A preliminary version of the requirements, to be developed further during the development process. Reaching the requirements will require sufficient analysis of the system and many features of the system need to be studied (7.2.2.5): safety functions, configuration and architecture, hardware safety requirements, software systematic capability requirements, capacity and response time, equipment and operator interfaces, including foreseeable misuse. Clearly, at this stage a lot of this is unknown, so the requirements need to be built during the development increments.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.2.2, figure 4 box ref 10.1
Validation plan for software aspects of system safety (M)	The plan presents the generic principles and main tasks, but not the details. This is the proper phase to outline the generic principle of validation (is it expected to be manual or automated, analytical or statistical; acceptance criteria, etc…) For safety-critical development, validation needs to be as efficient as possible and a good plan and development of validation processes helps in that.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.3.2, figure 4 box ref 10.2
Start configuration management (M)	Start configuration management, with tracing to design, verification and validation and documentation to ensure that all software items are identified and controlled during the process.	IEC 61508-3, 2nd ed., sub-clause 6.2.3
Start quality management	Start all the practices that ensure that the necessary practices are followed through the project	
Review readiness to start development (M)	A review is required to assess that all requirements are met.	

Other tasks:

- Studies of the target business, processes and work. Any studies of the target system so that we understand the business, the technology around our system. This in independent of the technology of our system. As this can be time-consuming, it needs to be carried out outside the increments. This can also be carried out in the context of the concept design task.
- Project planning for development of the software system. Planning based on the concept, the known development tasks and required validation tasks. Creation of a project plan.

- Verification plan. This will mostly consist of a Master Test Plan, which presents an understanding of how testing will be carried out in the process. The plan in generic and does not concern technology-related details. It will present the approach to testing, description of roles, basic processes (such as unit testing, integration testing, system testing).

10.5 The first increment – a planning increment

The first increment of the agile process is usually very special, because in it, the foundations for the development are created. During the increment, software architecture is planned that development can build upon. In the increment we take aim at the very first software version and the very first customer release. For those, we plan and decide, which product features or components should be developed first, in order to give maximum value to the customer and to enable fast feedback on the most essential things – and all this forms the basis for the development of the safety system.

Goals:

- Elaboration of the concept.
- Understanding what is important in it for the customer and what are the critical safety issues.
- Understanding the use of the system.
- Formulating a rough architecture, based on the “form” of architecture, not its implementation details.

Inputs:

- Outputs from start-up phase.
- New information from customer, product management, marketing, safety information, etc. sources.

Outputs:

- Updated concept.
- Updated project plan, safety plan, and validation plan.
- Updated safety requirements.
- Documented product usage.
- Hazard and risk analysis of use.
- Description of architecture.
- Analysis of architecture.
- Updated release plan (some description of the first plan).

IEC 61508 tasks

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Functional architecture design (M)	Design of the basic functional software architecture. The design should present the basic “form” of architecture, but not lock technologies, which will be chosen later. In agile development it is important o make a clear distinction between safety-critical and non-safety-critical architecture elements and to try to group them so that revalidation requirements apply to as few elements as possible, making dynamic, iterative development of those as flexible as possible.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.3, figure 4 box ref 10.3
Safety architecture design (M)	Design of the basic software safety architecture. The design should present the basic “form” of architecture, but not lock technologies, which will be chosen later.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.3, figure 4 box ref 10.3
Update of software safety requirements specification (M)	Based on any new designing, either on architecture or anything else, the collection of safety requirements needs to be updated.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.2.2, figure 4 box ref 10.1
Safety assessment of the basic architecture (M)	Assessment of the architecture about its safety. As the design of architecture elements is not done, this is a preliminary assessment, to be updated later.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.7.2, figure 4 box ref 10.6
Safety assessment of user interface concept (UI implemented in software)	Design or selection of UI concept is part of this phase and its safety needs to be assessed with analytical methods that help identify for example, human errors.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.7.2
Software architecture integration test plans (M)	Testing of the architecture is planned. As we have a plan of what the system might consist of, we can formulate a plan about how the integration will be carried out. This will apply to both testing integration of software into the hardware system and integration of the software system elements together. This plan will evolve later.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.3, figure 4 box ref 10.3
Selection of development tools (M)	Selection of languages, compilers, APIs, data formats etc. Usually there is an understanding of those already, but in this phase, the tools are finally chosen based on known needs. The tools need to be previously validated so that we know what accepted tools are available.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.4, figure 4 box ref 10.3
Selection of coding standards (M)	Selection of any coding standards would usually mean that a coding guideline is selected for the project. Any guidelines need to be developed before a project is started.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.4, figure 4 box ref 10.3
Review that architecture meets requirements and that all tasks have been carried out properly (M)	A review is required to assess that all process and project requirements are met.	IEC 61508-3, 2nd ed., sub-clauses 7.1.4.3, 7.1.4.7

Other tasks

- Updated user and business studies based on new information.
- User interface concept design. The basic principles and key elements of the user interface. Validation of the UI concept can be based on analysis and testing with prototypes. A safety validation is also required by IEC 61508.

10.6 Regular increments (several)

The number of “regular” increments can vary. In essence they are all of a similar nature. The important difference is whether they aim to produce a release candidate to a strict validation and release process or if the increment only aims at an increment for internal evaluation (including usage in a simulator and other controlled environments.

In agile development we need to assess very carefully how the proposed new features change the system concept and what hazards they might cause. This hazard analysis is best carried out for a set of features and not at a level of single features.

Goals:

- Creation of a new software version that produces more value.
- Keeping the software concept solid, yet evolving within the constraints and scope of the project.
- Addressing any need or impulse to changes in an open and positive attitude.

Inputs:

- Outputs from the previous increment.
- New information about need to make changes.

Outputs:

- New software version that provides more value and contains new features or improved performance or quality.
- Possibly: a release candidate for the release process.
- Evaluation of the new version – understanding of the product.
- Updated concept.
- Updated project plan, safety plan, and validation plan.
- Updated safety requirements.
- Updated hazard and risk analyses.
- Updated safety documentation.
- Updated release plan (some description of the first plan).

IEC 61508 tasks

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Update of hazard and risk analysis (overall system) (M)	Selection of a set of new features and a hazard analysis of the new, planned system version form the essential starting point to an increment.	IEC 61508-1, 2nd ed., Ch 7, Table 1
Update of software safety requirements specification (M)	Based on the selection of new features to develop, the collection of safety requirements needs to be updated.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.2.2, figure 4 box ref 10.1
(Updates to any plans and specifications made earlier) (M)	Most importantly: continuous updating of any safety requirements and other safety-critical information	
Low-level software integration testing (M)	Low level integration testing is often done continuously or, if that is not practical, daily or by other frequent cycle. Low-level integration testing is usually quite technical and does not verify most safety requirements.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.8, figure 4 box ref 10.3
High-level software integration testing (M)	Higher-level integration tests can verify that all software subsystems meet their requirements and behave as planned. This can consist of integrating the work of many teams. This testing can often start in an emulator.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.8, figure 4 box ref 10.3
Electronics integration tests or tests in the target machine (M)	Testing of integration of software and programmable hardware, that is, a device or a machine.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.5.2, figure 4 box ref 10.4
Correcting errors found in integration tests	An impact analysis needs to be carried out for the corrections of errors. In agile development, error corrections are often formulated as development tasks and thus follow the process of developing new features.	IEC 61508-3, 2nd ed., sub-clause 7.5.2.8
Planning of software operation and modifications procedure (M)	Writing of any needs instructions for using and modifying the software being developed.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.6.2, figure 4 box ref 10.4
Software verification	Testing of evaluation the software in many phases and many environments. Based on the Master Test Plan and detailed sub-plans for any testing activity used.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.9.2
Software functional safety assessment (M)	Assessment of functional safety of the new functionality. Often carried out with a FMEA analysis, but analysis methods can vary.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 8

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Software aspects of system safety validation (internal, unofficial) (M)	Validation that the total system complies with requirements at the intended safety integrity level. Testing is the main validation method. Analysis and modelling can supplement it (they will produce information to base testing on). Note: During the primary software development process, validation is only internal and informal and will reach an understanding of whether the software could be offered as a release candidate for customers or for the mass market. The external, official validation will be performed in the release process. However, the internal validation activities should usually be documented as well as the external, independent validation.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.7.2, figure 4 box ref 10.6
Review that all tasks have been carried out properly (M)	A review is required to assess that all process and project requirements are met.	IEC 61508-3, 2nd ed., sub-clauses 7.1.4.3, 7.1.4.7

Other tasks

- Assessing the project's situation.
- Planning and selection of new features for development, based on customers' needs, prioritisation using multiple criteria, release plan and other information.
- Risk analysis of new features.
- Impact analysis of new features.
- Design, implementation and verification of new features.
- Hazard and risk analysis and safety assessment of new features and the changed system.
- Regression testing of the system.
- Updating product documentation.
- Updating user documentation.
- Updating safety documentation.
- Updating project documentation.
- Evaluation of the increment; process and the resulting product.

10.7 Development of single feature

In agile development, development work is usually divided into small items, often features, but obviously the item can be any task that can be seen as individual effort.

This single feature will receive individual tracking as it proceeds through requirements, impact analysis, design, implementation and verification and safety assessment, with each step being suitably documented.

Note that corrections of errors found in integration testing are handled as “modifications”. In agile development error corrections are often defined as work tasks that are no different than developing new features. Thus, they can follow the same process and they need to meet most of the requirements. In fact, in agile development the whole development process can be seen as consisting of modifications.

Goal:

- A new feature that provides true value to stakeholders is successfully added to the product.

Inputs:

- Selection of items for development in this increment. One of those being this feature.
- The overall concept of the system, understanding of what the customer needs and what the system is used for.
- Understanding of what value this feature gives to customer and all other stakeholders.

Outputs:

- A developed feature, which has been verified to meet its requirements.
- Understanding of how the new feature impacts the system.
- Planning and design documents and artefacts.
- Updated product configuration.
- Assessed safety of the new feature.

IEC 61508 tasks

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Detailed design and development of a single feature, module, component or similar (M)	The basic task of designing and implementing of a single software item. IEC 61508-3 specifically states that the software shall achieve modularity and testability and capability of safe modification (7.4.5.4). Special testability reviews are in the agile context to an extent replaced by continuous collaboration with testers.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.5, figure 4 box ref 10.3
Impact analysis of the new feature (M)	Impact analysis is required for software modification. In agile development, the whole development process can be seen as consisting of modifications.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.8.2.3

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Detailed design and development of the software (M)	This is incremented from the designs of individual items.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.5, figure 4 box ref 10.3
Test specification for the software (M)	This is created incrementally from test designs of individual items. The specifications will apply to all test levels. Usually developers create module test specifications incrementally. Integration test specification can be the responsibility of an engineer or team responsible for integration and system test specification is the responsibility of a tester. All these are created in the context of a Master Test Plan or similar.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.5, figure 4 box ref 10.3
Detailed code implementation (M)	The everyday programming.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.6, figure 4 box ref 10.3
Code reviews (M)	Manual code review at the start of development, later, utilisation of automated code checking in the build system.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.6, figure 4 box ref 10.3
Module testing (M)	Module testing, usually by developers. Test-driven design and test frameworks produce the required documentation. Module tests are often re-run during low-level integration, giving more visibility to the process. Note that a “precisely determined testing configuration” is required (7.4.7). Automated unit tests (that are recommended for agile development) that are run in every build can assure that the configuration is not changed after tests have been run.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.4.7, figure 4 box ref 10.3
Software verification	Testing of evaluation the software in many phases and many environments. Based on Master Test Plan and detailed sub-plans for any testing activity used. Sub-clause 7.9.2.7 of IEC 61508-3 lists the required verification activities. The project needs an understanding at the project’s beginning that these activities need to be performed at some time. The Master Test Plan is a tool for sharing that understanding. But the detailed test plans and designs will be created at the time of design and implementation. A most critical aspect of verification is basing it to requirements, architecture and specifications, which means that there needs to be full tracing between all levels of specifications and tests, and a thorough review of test coverage against design elements.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.9.2

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Static verification of code	The software code needs to be statically verified that in conforms to specifications and coding standards and the validation plan. Manual verification is required at the start of development and tests suffice after that. However, automated static analysis, for example, during an automated integration process, is recommended. That is usually seen to be part of good agile development culture.	IEC 61508-3, 2nd ed., sub-clause 7.9.2.12
Software functional safety assessment	Assessment of functional safety of the new functionality. Often carried out with a FMEA analysis, but various other analysis methods can be used.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 8

Other tasks

- Assessing all non-safety related criteria of the feature.
- Writing of release notes or similar condensed documentation to tell others about the new feature and its state.

10.8 Release process activities

During a release process, a release candidate provided by the development process is assessed and validated and “packaged” into a form that it can be delivered to a customer or can be marketed even in mass-markets. The actual tasks depend on the company’s productization, delivery and product management practices.

The number of released products will also depend on the SIL level and required tasks – on lower SIL levels the validation procedures are lighter and thus it is more realistic to have several releases during any given period.

Goal:

- To assess whether a release candidate is suitable for releasing.
- Gaining validation for the product.
- To be able to deliver the product successfully to the customer.
- To be able to deploy the product successfully in the customer’s production system.

Inputs

- A developed product or system.
- Its documentation as produced by the development team.
- All assessment, verification and validation records.
- All quality management records.

Outputs:

- An independently validated product or system.
- Customer-grade documentation.
- Safety manual.
- Assurance that all required tasks have been performed during development.

IEC 61508 tasks

IEC 61508, 2nd ed. tasks during phase	Guidelines and task outputs	Standard reference
Software aspects of system safety validation (internal, unofficial) (M)	Validation that the total system complies with requirements at the intended safety integrity level. This is usually done by carrying out safety and reliability assessments, using many methods (FMEA being just one of those). Note: At this phase, the validation is done by an independent party, the independence of which can vary, based on the product's SIL level from an independent internal validator to an external organisation.	IEC 61508-3, 2nd ed., Ch 7, Table 1, sub-clause 7.7.2, figure 4 box ref 10.6
Assuring that the development process meets all requirements of applicable standards (M)	This means that we check that the requirements outlined in previous chapters have been fulfilled properly.	
Review that all tasks have been carried out properly (M)	A review is required to assess that all process and project requirements are met.	IEC 61508-3, 2nd ed., sub-clauses 7.1.4.3, 7.1.4.7
Safety manual	Product level and any necessary component level safety manual.	

Other tasks

- Adding to the product's documentation.
- Any productization activities.
- Additional verification testing that is not seen as the team's task.
- Validation of conformance with many other standards than IEC 61508 series.
- Internal and external training.
- Integrating the developed product fully in the product management system.
- Starting of support services for the new release.

10.9 Project closure activities

This is the part where a project is closed, based on a decision, turning the product into maintenance mode, changing responsible parties and persons. This part of project's lifecycle need not be influenced a lot by the nature of the development phase, except for one thing, which is freedom for timing. The agile process enables a company to more freely determine whether a product still needs development after a certain release, of whether the resources should be targeted on the development of other products. This is agile product development management, which helps a product company to be more agile on business level, not just projects.

In an agile multi-release process, all storing of documents, for example, concerning each validated release, have been done. Now, the most important thing is just to assess the project and release its resources.

11 Illustrative example: Automated trash collector

This section contains an illustrative example of the previously outlined approaches to agile development. The example is fully imaginary. It aims to make the process ideas presented more concrete. **Note that the description is not by any means complete, but just presents the most essential elements of agile safety-critical development**.

Concept: Automated trash collector

An automated device that collects and separates trash from streets, parks and city squares etc., used every day and after parties, concerts and other gatherings.

Figure 23. Conceptual image of the trash collector.

Release plans:

(1) First make a version that can be used when the area is separated, allowing for development of the collection technologies and understanding the issues. This is not meant to be sold, but provided to some large customers for evaluation. (2) Then, make a version that can be let to work among people. This should be a fully marketable product. And (3) finally, make a version that can detect people and animals and based on that information be more efficient.

The releases are created with a typical agile process with an increment flow (of yet unknown number of increments) from which the releases are made at suitable times – when it is seen that a release is sensible. During the planning phase, no dates have been set.

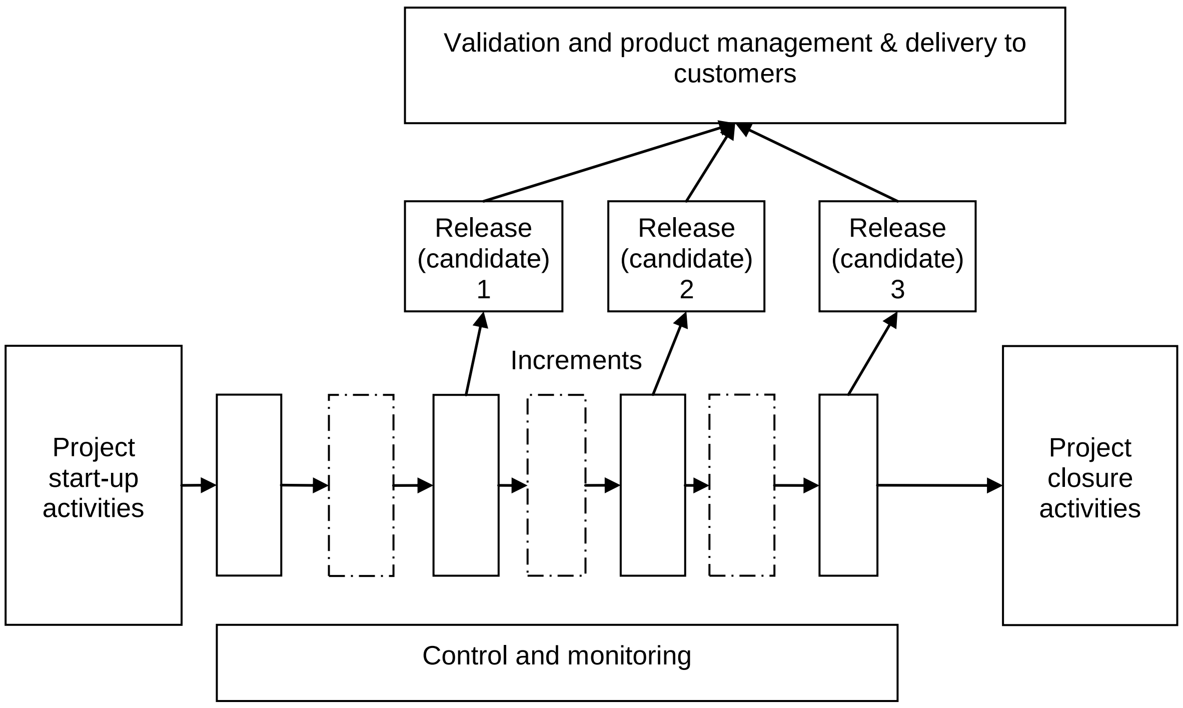
Validation and product management & delivery to customers
Release (candidate) 1
Release (candidate) 2
Release (candidate) 3
Increments
Project start-up activities
Project closure activities
Control and monitoring

Figure 24. The basic agile process used in the project.

Hazards and risks:

A general list of hazards: people getting hit with the device. Animals getting picked. People sitting on the device and falling down. The device getting on the driveways and blocking cars or emergency vehicles. A danger to people's legs.

The hazard list evolves through the development project as more is learned about the product and its use. At least three formal hazard analysis sessions are held and hazard related information is updated each time it appears during development.

Simplified agile development process:

As start-up activities the concept is visualised and its feasibility analysed. Hazards are assessed in the development team in an analysis session lead by an in-house hazard analysis expert. SIL level is defined to be 1, as this device has a slow speed and has mostly the potential to cause only minor damage. But it is understood that the SIL level needs to be reassessed as development continues.

Plans are made:

- The release plan is decided to be as above. All steps are essential to understanding how the next steps should be reached. The plan does not consist of a lot more detail than the concepts for the releases.
- The safety plan is written.
- Basic safety requirements are written for release 1. They consist of various requirements for operator safety, the maximum speed needs to be very limited, operation should be done from behind, emergency stop devices need to be located in all corners, the device must stop if hits something with a given force etc.
- The project plan is written.
- The validation plan is written. Validation is done in-house by the company's validation unit, which does not participate in the development.
- Verification plan is easy to create, as it is mostly compiled from references to the company's quality instructions.

The development team consists of hardware engineers, an industrial designer, software designers, a safety expert, a product manager and a trash collection expert hired from a customer. The team is lead by a project manager.

Steps until the first release

As the first increment, the basic architecture of the system is devised. For safety systems, traditional sensors etc. are used for detecting hits, monitoring speed etc. The safety of the proposed design is assessed. After the increment, the product is visualised more in detail, its manufacturability and implementability are assessed. Its use is simulated with drawings. The hazard analysis is revised for some details.

For the development increments, tasks are divided between the hardware people and software people:

- The hardware people develop the first trash picking devices (we will worry about storage later).
- The software people devise algorithms for detecting various kinds of trash. These are functional requirements.
- Safety device sub-team starts designing the safety system.
- A separate team is assigned to figuring out how living things can be detected - both people and animals, even though this technology is not needed yet. This development is guided separately from the other developments.
- The sub-teams get a list of tasks to do for the next 30 days which was selected to be the duration of an increment

Each evening, a short meeting is held together to see how things are progressing. It is held sitting down, contrary to some agile methods that hold stand-up meetings, because sitting signifies that we must take our time – let's stand up and run only when the system is safe!

During the course of designing, preliminary designs are developed by discussion, but at some point the designs are analysed with failure analysis. Mechanical designs are subjected to strict assessment for meeting design standards.

At the end of the first increment a prototype is crafted, which allows the team to assess the technology and to use the primitive user control panel (just start, stop, lock etc.)

The prototype is assessed as a whole for safety, and some design changes are added to the task list of the next increment

Note that this is the change management style for unfrozen designs: just agree the change, document it, and make the necessary checks and analyses later. The process called "change management" is not used in this phase as it is based on a different kind of thinking.

During the increment software verification happens as usual: code reviews, and testing. For testing, a simulator has been built where various events can be fed into the system. A software based system simulation model has been considered, but at the start, events are generated manually. Simulator developments are saved for later increments.

The next increments continue on the same track. During the last increment for the first release, the process shifts into a more formal mode. Designs are frozen and analysed fully. All design tasks are logged and documented. Verification tasks are documented. Errors are logged.

At the end of the increment, once again the product is assessed. This time the project has produced a working device which can be run in guarded area. Also, the required documents have been drafted, including the safety manual.

The device is assessed for its readiness to be moved into the release process and it gets the green light for that.

Also, the development team assesses its practices in a lessons learned session and decides to build a Wiki page for sharing information that should be kept and to restructure information in the development information system so as to make information more accessible.

Now, a small separate team takes the product and leads it through the release process which is divided into two "tracks":

- Validation. This is done according to the validation plan and by the internal validation unit.
- Productization. The product is readied as a supportable product. Manuals are finalised, stickers are made, support systems are initialised, trainers are trained, etc. This is done even though the first release is only distributed to some key customers.

Meanwhile, the development team can continue into the next increment.

Steps to the second release

The first release has provided the manufacturer a solid product, which is now developed further. It has provided new knowledge, for example, that the performance of the device is based on its speed and size of the collector openings. Both are clearly safety issues.

The next goal is to make a version that can be left to work among people.

As the concept is now changed from the original plans and the device's use is understood more fully, a new hazard analysis is made. It provides new safety requirements, such as: adjustable modes of operation (run by operator or autonomous operation; both of which can utilise different settings for speed and the trash collection devices; locking of controls is required so that people cannot tamper with it; the device must stop especially fast if it hits something). Giving information to the operator if the device is stuck is classified as a functional requirement and a non-mandatory extra. The others are safety requirements and require validation.

As before, the teams create a task lists, synchronising the hardware and software issues so that a working prototype can be built at the end of each increment. We do not yet know how many are needed, but "guestimate" that six or seven might be needed. The team uses a browser-based task management system to control the tasks.

The user interface tasks are understood to be important in this task, so the industrial designer has high-priority tasks in the designing of an operator panel.

Experiences from the first release have helped the team to have some understanding of how much work they can carry out during each increment.

The teams then get to work.

After the first increment the operator panel is well designed and can be subjected to usability assessment. Human errors are analysed using a special analysis technique. End user tests are carried out on the factory floor, well guarded, using the prototype. Not all functions work yet, but they are simulated. The manufacturer's own people act as test persons as well as the key customer's representative who is participating in the project. But he knows the product now too well to be a reliable usability test person.

Work on this release goes to release process as with the first release.

As safety-critical and functional requirements and modules have been clearly separated, it is easy to show to the validator which items have been changed, what impacts the changes have and how the impacts have been verified to be in control. Thus, the re-validation is an easy, quick and cost-effective process.

Steps to the third release

Again, the team has learned a lot from the first two releases and the technical issues are now fully understood and the product provides plenty of value to customers. Still, a basic hypothesis in the beginning of the development project was that if the product can detect living things, it could be made to work a lot more effectively.

Now that there has been a lot of experiences, that thought is reassessed. Remember that this was as demanding research task that was assigned to a separate team, which utilised agile methods, but was not linked into the main incremental process – until now.

Tests have, however, shown that the detection is not as accurate as would be required for a mass-market product and thus the idea is not dropped.

But during the development of the second release, new ideas have been collected which are now assessed. Mostly they consist of logistics: keeping track of how full the container is and reporting it to the operator. This is not a safety-critical functionality, so the third release can be implemented with a reconfiguration of the functional user interface and even revalidation is not required.

The third release demonstrates that plans change – old plans are dropped and new plans emerge. But still, the original release plan was useful: it provided the team with a sense of continuity and chain of new developments.

12 Guidance for changing the process into more agile

12.1 Prerequisites

Shared sense of a need to change

Agile is not a goal as such, but a means to meet some business or other goals.

For any change in an organisation, there should be a clear, shared sense of urgency to change processes and a feeling that the current way of action is not sufficient anymore. Without that, the probability of success is not high enough. You need to want to change. A requirement for that can be formulation of a business case for process development: what benefits would a different release practice bring. Everyone should feel that there are shared problems that the agile development model would solve.

Solid process understanding and vision

There should be an understanding of software and product development processes – agile and otherwise, which should help to understand one basic question: do we really aim at maximum agility or a balanced approach. This cannot be repeated too often: the goal is not to be agile, but get the best results by using processes that best fit the goals in this particular environment. (As orientation material on balancing agile and plan-driven methods we recommend Barry Boehm's and Richard Turner's book "Balancing Agility and Discipline – A Guide for the Perplexed".)

Professionalism in action

The traditional approach to process change is that all the pieces are not in place, and an attempt to change process may result in chaos and failure. So, before transformation the organisation needs to be able to fulfil, for example, all the requirements of IEC 61508 or other applicable standards with no problems. That provides a baseline for process development and makes it possible to see clearly any process problems.

The quality of current activities should be high and the execution of current process professional. If this is not the case, the change probably should not be attempted.

Experts and collaboration

Expertise on agile is needed in the transformation, to guide the rest of the people in the transformation process. Process people or consultants are needed who can lead the transformation. Here it should be noted that agile consultants do not necessarily understand all the process requirements of safety-critical development and the leading principles of safety and risk management. Even world class agile experts may have a limited understanding of testing and verification and validation. All the necessary expertise areas are required to be presented in the process.

12.2 Ten generic guidelines for process development

These are seen as important guiding principles for the transformation of traditional process models into more agile:

- Respect your own engineering skills and experiences in the process development.
- Do not follow fashion. Seek proven practices and analyse their suitability in your environment and culture.
- Understand that not everything needs to be agile and that “more agile” will not mean “better”.
- Remember that the more safety-critical the system and its development are, the more control is required for the process and that will usually mean less agility.
- No single paradigm is sufficient. Being just agile or just plan-driven is not enough. Agility in some areas may need less agility in others, for specific process needs and for balance.
- Much of any company’s current activity is tacit in nature and not formulated or documented. Thus, you might be blind to your current success factors. External consultants are often needed to see the essentials in your activities.
- Safety-critical development is all about managing risks. In the same way, you need to manage the risks of making process changes.
- Find the style that is best for you, is the message in all development. If you act like everyone else, how could you be the best in your branch?
- Process developments are never just mechanistic process issues. Agility needs to fit into the corporate culture, which is a hard thing to change.
- Co-operation and collaboration between all stakeholders is one key issue in Agile. The same applies to process development.

12.3 Risk analysis of the transformation

A risk analysis should be made for the process change in order to identify potential pitfalls. This author has previously devised a risk map to present risk areas to assess. It is based on the idea that a transformation of project practices is very much a cultural change. Pure processes are easy to change, but in software development it is a question of how people work together.

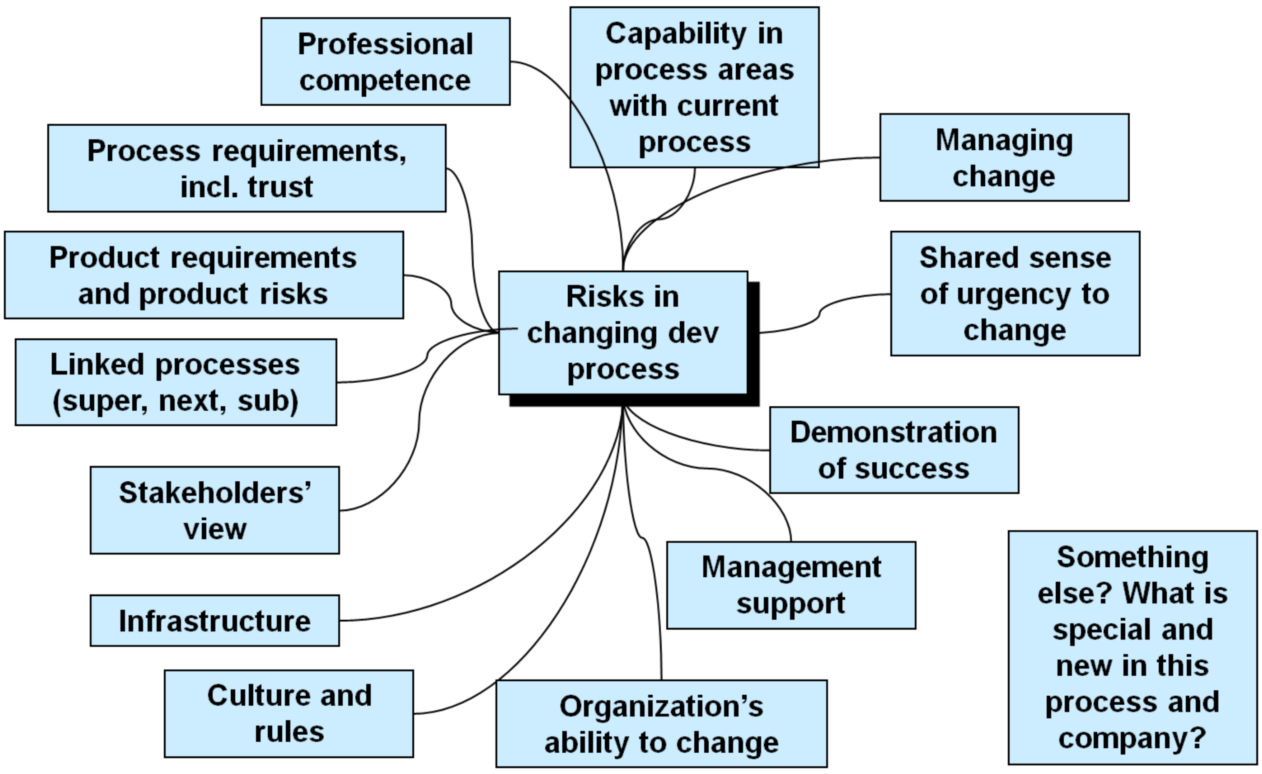
Professional competence
Capability in process areas with current process
Process requirements, incl. trust
Managing change
Product requirements and product risks
Shared sense of urgency to change
Risks in changing dev process
Linked processes (super, next, sub)
Demonstration of success
Stakeholders' view
Management support
Something else? What is special and new in this process and company?
Infrastructure
Culture and rules
Organization's ability to change

Figure 25. Areas of assessment in the risk analysis (Vuori, 2010b).

A risk analysis of this kind is usually carried out in a risk analysis session, taking a couple of hours, led by a risk analysis consultant. Participants should include all stakeholders, including:

- Product development.
- Software development.
- Quality.
- Safety and risk management.
- Product business.
- Sales.
- Subcontracting.
- Management.

Elements of software process – practices, processes, techniques and tools – can be thought to form a toolbox of items that we are free to choose from. Therefore, there as many moments when we need to make a decision: shall we use that particular practice or not? Boehm and Turner (2003b) present a rule that is particularly well suited to safety-critical development:

> Is it riskier for me to apply (more of) this process component or to refrain from applying it?

12.4 Step by step strategy

Basic elements that provide value for any process model

One strategy is to at first implement some enabling practices and other process elements that bring benefits to any development process. These include:

- More thorough unit testing and code quality assurance so that there is a solid base for changing the product.
- Continuous integration so that there is always a working product at hand, to assess how it behaves and to learn from it.
- UI-level test automation. Automating as many as possible of the tasks that are required for verification and validation.
- Agile process automation – automated reporting tools and information systems that can easily be tailored to changing situations in a project.
- Automated tracing. When a requirement or design or implementation is being planned to change, or has changed, tools are needed to see immediately what it affects and what also needs to be addressed. Good information systems provide this functionality.

Make rules clear

Agile requires clear rules to be successful – freedom is always best within constraints. The following need to be defined:

- Who owns the products, the product business and safety?
- Who owns the development process?
- What are the principles of product development – is it customer-led or innovation-led? Both may require different processes and participation.
- How do we weight the stakeholders' opinions. Is product development a democratic process or is some party clearly leading it?
- How open do we really want to be towards the customers?
- How much do we really wish to engage subcontractors?

Communication and collaboration

As agile is about communication and people working together, constantly learning, one needs to create a proper environment for that to happen.

- More self-reflection is needed in the process. All participants need to look into the product and process frequently to see what in their way of action needs to be improved. This calls for frequent lessons learned / retrospect meetings in the process.
- Team building and improved leading of teams. Teams need to able to work as teams and learning that, and learning to lead teams in more collaborative ways, takes a time.
- Improved communications tools. Anything that helps people to communicate better, yet provides a peaceful environment for those tasks and for persons that will benefit from it.
- Improved communication between distributed sites (including any subcontractor sites). No party should be left in a secondary role in communication.

Make time-consuming phases efficient

Validation and especially external certification can be time-consuming and thus difficult to implement in the agile process. Therefore, the processes need to be developed so that validation and certification are as efficient as possible.

- Creation of external processes outside the incremental main development process.
- Clear understanding of at what times the validation related tasks need to be carried out.
- Making a very clear distinction between safety-critical and other requirements, and SIL levels, so that the elements requiring validation or re-validation can be identified exactly.
- Efficient information systems so that any paperwork or routines or finding and checking of development records do not take time.
- Flexible arrangements for validation. If external validators are used, the situation can be reassessed – could the validation be carried out internally?

Provide control

Create such process features that help in controlling and bringing possible problems to light as soon as possible.

- Split requirements into smaller items, aiming for similar size to aid in planning and estimation.
- Improve meetings – make them more frequent, get all people to participate; develop meeting cultures.
- Create dashboards and information systems that visualize the project's status to everyone.

Split the process

Any development process can be split into more independent iterations. Essential factors for those are:

- A chain of events from planning of what to do, to evaluation of what has been achieved.
- Aiming for a demonstrative whole at the end of each increment.
- True planning in the beginning of an increment.
- Utilising real product development practices in defining requirements. That is, the input should be something that provides value: new use case, user story or similar, but NOT a technical specification or a change request. Change requests belong in the domain of maintenance, not in the domain of new product development lifecycle.

Increase number of releases

Increasing the number and frequency of releases can be gradual, and in safety-critical development it should probably be so.

- Practice the making of release plans that are rougher and based on value and risk, not just technology and functions.
- The increment-based lifecycle can be exercised during implementing – select features to implement and test during an increment and in the second phase, transfer more design work inside the increments. At the same time, the practices of architecture design need to be developed so that it can also evolve sufficiently.
- Development of release processes and practices so that all steps leading to the customer's utilisation of the product are as efficient as possible, yet do not compromise safety in any way.
- When the processes are in good shape, releases can be added in a controllable manner if there is benefit gained from that.

Add any other practices that bring value

There are and will be many good practices in the agile culture. The idea is not to implement all of them, but only those that really bring value. For example, while pair programming – an important old agile practice – seems to have lost its appeal somewhat lately, it could be a tool in transferring practical knowledge and experience from older developers to new developer generation.

But nevertheless, any mature software development shall not restrict its views to one paradigm only. The current agile culture did not just emerge, but is a result of process evolution, combining ideas and practices into a new whole. In the same way, processes in organisations should evolve by making changes that provide quality or performance improvements to projects. Agile has sometimes been called a dumping ground for practices, and companies cannot afford that.

So, for new process development ideas one should look into many relevant areas, not just agility. The others include:

- New project development.
- Software science.
- Safety analysis and safety and risk management.
- Quality management.
- Testing.
- Information management.
- Communication.
- Process control.

Continuous improvement

Continuous improvement of the process. Keep improving the process and meeting any problems one at a time.

The agile processes applied need to be developed further, just like any processes. Processes are only optimal in a given context and learning changes that – not to mention any external influences from technology, changing standards and so on.

Continuous improvement should have an important role in any organisation.

13 Conclusions

There are many reasons why agile development could be especially beneficial for safety-critical development:

- Customers need time to understand all safety percussions and early releases allow for that.
- Communicating design and safety information is easier, as agile emphasises verbal communication, not just written documentation.
- If a new technology should prove to be unreliable, the situation can be detected early and changes be made.
- Risk analysis has a gradually evolving scope (the concept), which should improve its quality.
- Safety assessments made in small increments can be more focused, delivering better results.
- A rhythmic process of integration – at all levels, from code to customer production systems – should reduce many kinds of problems.
- Even if the increments are not targeted for production, their behaviour can be well tested, assessed and understood in simulation. Any corrective measures can be based on practical observations from executing the system, not just analysis.
- Agile can provide a good learning experience for all involved, if applied properly.

However, agile development may have some potential obstacles, including the high requirements for verification and validation, safety assessments and similar tasks that are required to approve a safety-critical system to market. Also, traditionally the agile processes have been mostly used in non-safety-critical situations and the outlook on processes has been very much different to what the safety-critical development requires:

- Priority on delivery <> priority to safety and reliability.
- Focus on customer "wants" <> focus on customer "needs" identified by careful analysis.
- Technical critical point implementation <> Technical critical point validation.
- Communication based on informality <> communication needs objective information about use and safety needs.

But the obstacles can be overcome with a good strategy and professionalism, and in many cases agile development can become a very fruitful approach to developing safety-critical software.

14 References

- Ohjelmaturva publications:

Paalijärvi, Jani. 2010. Development of Safety-Critical Software using Agile Methods. Master's Degree Programme in Information Technology. Tampere University of Technology. 54 pages.

Paalijärvi, J., Katara, M., Karaila, M. & Parkkinen, T. 2010. Agile development of safety-critical software for machinery: A view on the change management in IEC-61508-3. The 6th International Conference on Safety of Industrial Automated Systems SIAS 2010, Tampere, Finland, 14-15 June, 2010 6 p.

Vuori, Matti; Virtanen, Heikki, Koskinen, Johannes & Katara, Mika. 2011. Safety Process Patterns in the Context of IEC 61508-3. Tampere University of Technology. Department of Software Systems. Report 15. 128 p. Available at the Tampere University of Technology DPub system: http://dspace.cc.tut.fi/dpub/.

- Other references:

5th Annual State of Agile Development Survey. 2010. Final summary report. November 7, 2010. VersionOne. 33p. Available at: http://www.infoq.com/news/2010/08/state-of-agile-survey.

Bernier, Alex; Burger, Dominique, and Marmol, Bruno. 2010. Wiki, a New Way to Produce Accessible Documents. In: K. Miesenberger et al. (Eds.): ICCHP 2010, Part I, LNCS 6179, pp. 20–26, 2010.

Boehm, Barry & Turner, Richard. 2003a. Observations on Balancing Discipline and Agility. Proceedings of the Agile Development Conference (ADC'03). 8 p.

Boehm, Barry & Turner, Richard. 2003b. Balancing Agility and Discipline – A Guide for the Perplexed. Addison-Wesley. 266 p.

Cawley, Oisín; Wang, Xiaofeng, Wang and Richardso, Ita. 2010. In: Lecture Notes in Business Information Processing, 1, Volume 65, Lean Enterprise Software and Systems, Part 1, Pages 31-36

Cockburn, Alistair. 2007. Agile Development – The Cooperative Game. Second Edition. Addison-Wesley. 467 p.

Coplien, James. 2011. Agile 10 years on. InfoQ. This article is part of the Agile Manifesto 10th Anniversary series that is being published on InfoQ. [Referenced 2011-03-11] Available at: http://www.infoq.com/articles/agile-10-years-on.

Jacobsen, Björn & Norrgren, Marcus. 2008. Agile vs. Plan-driven in safety-critical development cases. A clash of principles? Masters thesis, Lund university, Department of Informatics. 194 p.

Coplien, James O.; Bjørnvig, Gertrud. 2010. Lean Architecture: for Agile Software Development. Wiley. 376 p.

Coplien, James. 2009. Balancing the tension between lean and agile? Presentation slides. Presented in Projektinhallintapäivä 2009 [Project Management Day 2009] at Tampere University of Technology. Available at: http://www.cs.tut.fi/tapahtumat/projektinhallinta09/JimCoplien_12082009.pdf.

Ge, Xiaocheng; Paige, Richard F. and McDermid, John A. 2010. An Iterative Approach for Development of Safety-Critical Software and Safety Arguments. Department of Computer Science, University of York, UK. IEEE Computer Society. Proceedings of the 2010 Agile Conference. Pp. 35 – 43.

IEC 61508 FAQ. Edition 2.0. [Referenced 2011-03-23] Available at: http://www.iec.ch/functionalsafety/faq-ed2/.

Kennaley, Mark. 2010. SDLC 3.0. Beyond a Tacit Understanding of Agile. Towards the next generation of Software Engineering. Fourth Medium Press. 280 p.

Kruchten, Philippe. 2011. The Elephants in the Agile Room. Philippe Kruchten's Weblog. Available at: http://pkruchten.wordpress.com/2011/02/13/the-elephants-in-the-agile-room/

Larman, Craig. Agile and Iterative Development. A Manager's Guide. Addison-Wesley. 342 p.

Leppänen, Marko. 2010. Literary survey: Using Acceptance Test-driven Development in Safety Critical Environments. Tampere University of Technology. Unpublished. 5 p.

Likert, Jeoffrey K. 2004. The Toyota Way. 14 management principles from the world's greatest manufacturer. McGraw-Hill Professional. 330 p. (Finnish title: Toyotan tapaan).

Manifesto for Agile Software Development. [Referenced 2011-03-01] Available at: http://www.agilemanifesto.org/.

Moser, Raimund; Abrahamsson, Pekka; Pedrycz, Witold; Sillitti, Alberto and Succi, Giancarlo. 2007. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team In: B. Meyer, J.R. Nawrocki, and B. Walter (Eds.): CEE-SET 2007, LNCS 5082, pp. 252–266, 2008.

Mustafa, Ally; Darroch, Fiona & Toleman, Mark. 2005. A Framework for Understanding the Factors Influencing Pair Programming Success. XP 2005, LNCS 3556, pp. 82-91.

Nonaka, Ikujiro & Takeuchi, Hirotaka. 1995 The Knowledge-Creating Company. How Japanese Companies Create the Dynamics of Innovation. Oxford University Press. 284 p.

Principles behind the Agile Manifesto. [Referenced 2011-03-01] Available at: http://www.agilemanifesto.org/principles.html.

Paige, Richard F.; Charalambous, Ramon; Ge, Xiaocheng and and Brooke, Phillip J. 2008. Towards Agile Engineering of High-Integrity Systems. Lecture Notes in Computer Science, 2008, Volume 5219, Computer Safety, Reliability, and Security, Pages 30-43.

Poppendieck, Mary & Tom. 2007. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley. 276 p.

Rohunen, Anna; Rodriguez, Pilar; Kuvaja, Pasi; Krzanik, Lech and Markkula, Jouni. 2010. Approaches to Agile Adoption in Large Settings: A Comparison of the Results from a Literature Analysis and an Industrial Inventory. In: M. Ali Babar, M. Vierimaa, and M. Oivo (Eds.): PROFES 2010, LNCS 6156, pp. 77–91, 2010. Springer-Verlag Berlin Heidelberg.

Rottier, Pieter Adriaan and Rodrigues, Victor. 2008. Agile Development in a Medical Device Company. Agile Conference 2008. IEEE Computer Systems.

Rowley, Darren & Lange, Manfred. 2007. Forming to Performing. The Evolution of an Agile Team. AGILE 2007.

Scharmer, Claus Otto. 2001. Self-transcending Knowledge: Organizing around Emerging Realities. In: Nonaka, Ikujiro & Teece, David (ed.). Managing Industrial Knowledge – creation, transfer and utilization. SAGE Publications. pp. 68-90.

SFS-EN-61508-1. 2nd edition, 2011-01-24. Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 1: General requirements. (Sähköisten/elektronisten/ohjelmoitavien elektronisten turvallisuuteen liittyvien järjestelmien toiminnallinen turvallisuus. Osa 1: Yleiset vaatimukset.) 117 p.

SFS-EN-61508-3. 2nd edition, 2011-dd-pp. Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 1: Software requirements. (Sähköisten/elektronisten/ohjelmoitavien elektronisten turvallisuuteen liittyvien järjestelmien toiminnallinen turvallisuus. Osa 3: Vaatimukset ohjelmistolle.)

SFS-EN 61508-4. 2nd edition, 2010-11-22. Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 4: Definitions and abbreviations. (Sähköisten/elektronisten/ohjelmoitavien elektronisten turvallisuuteen liittyvien järjestelmien toiminnallinen turvallisuus. Osa 4: Määritelmät ja lyhenteet). 67 p.

SFS-EN 61508-5. 2011. Functional safety of electrical/electronic/programmable electronic safety-related systems. Part 5: Examples of methods for the determination of safety integrity levels.

SFS-EN 62061. 2005. Safety of machinery. Functional safety of safety-related electrical, electronic and programmable electronic control systems. (Koneturvallisuus. Turvallisuuteen liittyvien sähköisten, elektronisten ja ohjelmoitavien elektronisten ohjausjärjestelmien toiminnallinen turvallisuus.) 1 + 197 p.

Maria Siniaalto, Maria and Abrahamsson, Pekka. 2007. Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study. In: B. Meyer, J.R. Nawrocki, and B. Walter (Eds.): CEE-SET 2007, LNCS 5082, pp. 143–156, 2008.

VanderLeest, Steven H. and Buter, Andrew. 2009. Escape the waterfall: agile for aerospace. Digital Avionics Systems Conference, 2009. DASC ’09. IEEE/AIAA 28th, 2009. Pp. 6.D.3-1-6.D.3-16.

Wikipedia. Agile software development. [Referenced 2011-05-08]. Available at: http://en.wikipedia.org/wiki/Agile_software_development.

Wikipedia. Code refactoring. [Referenced 2011-05-08]. Available at: http://en.wikipedia.org/wiki/Code_refactoring.

Wikipedia. Continuous integration. [Referenced 2011-05-08]. Available at: http://en.wikipedia.org/wiki/Continuous_integration.

Wikipedia. Exploratory Testing. Referenced 2011-03-14] Available at: http://en.wikipedia.org/wiki/Exploratory_testing.

Wikipedia. Fagan inspection. [Referenced 2011-05-16] Available at: http://en.wikipedia.org/wiki/Fagan_inspection.

Wikipedia. Kanban. [Referenced 2011-03-10] Available at: http://en.wikipedia.org/wiki/Kanban.

Wikipedia. Safety Integrity Levels. [Referenced 2011-03-14] Available at: http://en.wikipedia.org/wiki/Safety_Integrity_Level.

Wikipedia. Scrum (development). [Referenced 2011-03-15]. Available at: http://en.wikipedia.org/wiki/Scrum_(development).

Wikipedia. Test-driven development. [Referenced 2011-03-15]. Available at: http://en.wikipedia.org/wiki/Test-driven_development.

Wikipedia. The Toyota Way. [Referenced 2011-03-09] Available at: http://en.wikipedia.org/wiki/The_Toyota_Way.

Wilson, John & Whittington, Claire. 2001. Implementation of Self-Managed Teams in Manufacturing: More of a Marathon than a Sprint. AI & Society, issue15, pp. 58-81.

Woodward, Elizabeth; Surdek, Steffan & Ganis, Matthew. 2010. A Practical Guide to Distributed Scrum. IBM Press. 189 p.

Vuori, Matti. 2010a. Testaus ja ongelmanratkaisu – eli miten Leanin oppiva organisaatio pääsee eroon bugeista [Testing and problem solving – or how Lean's learning organization gets rid of bugs]. 9 p. Available at: http://mattivuori.net/julkaisuluettelo/liitteet/testaus_ja_ongelmanratkaisu.pdf.

Vuori, Matti. 2010b. A risk map is an important tool in 21st century risk analysis practice. A collection of risk maps. A slide set. Available at: http://www.mattivuori.net/julkaisuluettelo/liitteet/collection_of_risk_maps.pdf.

Appendix 1: The 14 principles of The Toyota Way

From Wikipedia article "The Toyota Way": http://en.wikipedia.org/wiki/The_Toyota_Way. Used under the license CC-BY-SA, see http://creativecommons.org/licenses/by-sa/3.0

Section I – Long-Term Philosophy

Principle 1: Base your management decisions on a long-term philosophy, even at the expense of short-term financial goals.

- People need purpose to find motivation and establish goals.

Section II – The Right Process Will Produce the Right Results

Principle 2: Create a continuous process flow to bring problems to the surface.

- Work processes are redesigned to eliminate waste (muda) through the process of continuous improvement — kaizen. The seven types of muda are: 1. Overproduction, 2. Waiting (time on hand), 3. Unnecessary transport or conveyance, 4. Overprocessing or incorrect processing, 5. Excess inventory, 6. Motion, 7. Defects

Principle 3: Use "pull" systems to avoid overproduction.

- A method where a process signals its predecessor that more material is needed. The pull system produces only the required material after the subsequent operation signals a need for it. This process is necessary to reduce overproduction.

Principle 4: Level out the workload (heijunka). (Work like the tortoise, not the hare).

- This helps achieve the goal of minimizing waste (muda), not overburdening people or the equipment (muri), and not creating uneven production levels (mura).

Principle 5: Build a culture of stopping the production line to fix problems, to get quality right *the first time.*

- Quality takes precedence (Jidoka). Any employee in the Toyota Production System has the authority to stop the process to signal a quality issue.

Principle 6: Standardized tasks and processes are the foundation for continuous improvement and employee empowerment.

- Although Toyota has a bureaucratic system, the way that it is implemented allows for continuous improvement (kaizen) from the people affected by that system. It empowers the employee to aid in the growth and improvement of the company.

Principle 7: Use visual control so no problems are hidden.

- Included in this principle is the 5S Program – steps that are used to make all work spaces efficient and productive, help people share work stations, reduce time looking for needed tools and improve the work environment. Sort: Sort out unneeded items, Straighten: Have a place for everything, Shine: Keep the area clean, Standardize: Create rules and standard operating procedures, Sustain: Maintain the system and continue to improve it

Principle 8: Use only reliable, thoroughly tested technology that serves your people and *processes.*

- Technology is pulled by manufacturing, not pushed to manufacturing.

Section III – Add Value to the Organization by Developing Your People

Principle 9: Grow leaders who thoroughly understand the work, live the philosophy, and teach it to others.

- Without constant attention, the principles will fade. The principles have to be ingrained, it must be the way one thinks. Employees must be educated and trained: they have to maintain a learning organization.

Principle 10: Develop exceptional people and teams who follow your company's philosophy.

- Teams should consist of 4-5 people and numerous management tiers. Success is based on the team, not the individual.

Principle 11: Respect your extended network of partners and suppliers by challenging them and helping them improve.

- Toyota treats suppliers much like they treat their employees, challenging them to do better and helping them to achieve it. Toyota provides cross functional teams to help suppliers discover and fix problems so that they can become a stronger, better supplier.

Section IV – Continuously Solving Root Problems Drives Organizational Learning

Principle 12: Go and see for yourself to thoroughly understand the situation (Genchi Genbutsu).

- Toyota managers are expected to "go-and-see" operations. Without experiencing the situation firsthand, managers will not have an understanding of how it can be improved. Furthermore, managers use Tadashi Yamashima's (President, Toyota Technical Center (TTC)) ten management principles as a guideline: 1. Always keep the final target in mind. 2. Clearly assign tasks to yourself and others. 3. Think and speak on verified, proven information and data. 4. Take full advantage of the wisdom and experiences of others to send, gather or discuss information. 5. Share information with others in a timely fashion. 6. Always report, inform and consult in a timely manner. 7. Analyze and understand shortcomings in your capabilities in a measurable way. 8. Relentlessly strive to conduct kaizen activities. 9. Think "outside the box," or beyond common sense and standard rules. 10. Always be mindful of protecting your safety and health.

Principle 13: Make decisions slowly by consensus, thoroughly considering all options; implement decisions rapidly (nemawashi).

- The following are decision parameters: 1. Find what is really going on (go-and-see) to test, 2.Determine the underlying cause, 3. Consider a broad range of alternatives, 4. Build consensus on the resolution, 5. Use efficient communication tools,.

Principle 14: Become a learning organization through relentless reflection (hansei) and continuous improvement (kaizen).

- The process of becoming a learning organization involves criticizing every aspect of what one does. The general problem solving technique to determine the root cause of a problem includes: 1. Initial problem perception, 2. Clarify the problem, 3. Locate area/point of cause, 4. Investigate root cause (5 whys), 5. Countermeasure, 6. Evaluate, 7. Standardize

Appendix 2: A glossary of agile (and some Lean) terms

This is a glossary of agile terms designed to aid people's understanding of the special terms used in agile culture. The glossary doesn't just repeat common definitions, but aims to give some guidance and reflection on the terms meaning. The descriptions of the terms are based on various sources, as there are no standards available for the terms. Many glossaries are written, for example, to support one agile method only, which is not sufficient for a general use of the terms in process development – not for today's needs and certainly not for future needs.

This glossary doesn't restrict itself in any particular agile method. Any particular method may have stricter, broader or otherwise different definitions.

The glossary is mainly aimed to support software development and so, when "software" is mentioned, the definition might as well mean "product" or "system" in another context.

Generic, well known software development terms are omitted from this glossary unless they have a special meaning in the agile context. The same applies to project and quality management terms.

Wikipedia references are preferred here, as they are neutral, immediately available, cross-linked to other concepts and evolving – as any glossary also should be.

Note: This glossary should be seen as a draft only as it was created rapidly during the spring of 2011 for the needs of Ohjelmaturva project. Development of glossaries should usually be a long-term project and therefore this glossary will be further developed later.

Term	Definition	Reflection from a traditional point of view
A		
Acceptance testing	This is a generic term referring usually to the customer doing testing that validates that the product is suitable for the agreed purpose and that the product meets all requirements. In agile software development, the term is also used in smaller scale to refer to tests done inside the development team that demonstrate that the implementation has been made according to customer's requirements. Especially in the case of information systems, this does not replace the larger scale, independent customer tests.	This is a very relevant concept in any development process when it includes customer acceptance. Note also that in many processes there may be internal acceptance tests which should not be confused with customer-driven testing.
Acceptance test driven development	See: "test-driven development", "acceptance testing"	
Agile software development	Development of products, systems and software in "agile" way, meaning usually: a) Being based on the principles of Agile Manifesto. b) Using an incremental development lifecycle with quite short increments (from one week to some weeks, instead of months). c) A development process where requirements evolve through the process and change is welcomed. Often, self-organising cross-functional teams are associated with agile software development, but that is mostly a cultural pattern and not a requirement for being agile.	In traditional development, pure waterfall processes have rarely been used. More often than not project are incremental in nature and requirements evolve through the process. The main difference is that the process sees changing requirements as a problem, and not a positive thing.
Agile practices	The practices used in agile software development. A particular development process can use some or several, but not necessarily all of the practices. The practices include: user stories, cross-functional teams, unit testing, refactoring, continuous integration, burndown charts, pair programming and more.	Traditional processes also consist of many practices – CMMI process areas, various software engineering practices, quality techniques etc., which are chosen to be suitable for the development process and requirements.
Agile Manifesto	A declaration of values and principles of agile software development. See: http://agilemanifesto.org/	
Agile software development life cycle	A defined software development life cycle (SDLC) that is agile and utilises agile practices. An SDLC is not necessarily "all agile", but at some point it is seen sufficiently agile to be called such.	
Agile principles	See: "Agile Manifesto".	
Agile processes	The processes used in agile software development during the software development lifecycle. The processes are very much related to "practices". Agile software development is actually quite process-dependent but as it emphasised people and interaction, the term "process" is not used much, instead they are usually referred to as "practices".	In traditional development, processes are emphasised and form the backbone of the development lifecycle. They are also linked in various ways to the company's other processes and functions, such as sales to requirement specification, and testing to quality management.

Term	Definition	Reflection from a traditional point of view
Agile project management	Project management in the agile context.	Project management is project management and its very basic principles are the same in any development paradigm.
Agile software development	See: "agile software development".	
Analysis story	A special type of story that doesn't describe user requirements or activity, but an analysis task that produces information for the composition of the user stories. The term is used in such agile practices, where development items are seen as "stories" by that receive similar control. So, the term is based purely on the goal of keeping the project and process terminology concise and to harmonize control mechanisms by treating all kinds of tasks as (at least structurally) similar. See: "story".	In the non-agile development, this would simply be called more exactly "analysis task" or "study", which would not confuse the nature of the task.
Agile testing	a) Testing that is not based on strict pre-planned test designs, but reacts quickly to the actual state of the software to be tested. Often in the form of exploratory testing that emphasises simultaneous learning, test design and execution. While there are fewer plans than in pre-planned testing, agile testing can use plans, especially higher level plans describing the approach and environment, pre-selected tools etc. Rapid ad-hoc test automation is also considered agile testing. Agile testing can be combined with systematic testing in various ways. b) In a more generic meaning, testing in the context of an agile development process. See: http://en.wikipedia.org/wiki/Agile_testing, http://en.wikipedia.org/wiki/Exploratory_testing	Agile testing can very much be used in plan-driven development too. Sometimes it may be called ad-hoc testing and only supplements the systematic testing.
B		
Backlog	A list of "things to do" in a various context: A product backlog refers to user stories, requirements, functionality or just any task that has been collected and is waiting to be worked on. A project backlog refers to the same in the project's context. A sprint/increment backlog refers to the same inside an increment. The backlog is something that is decided upon when the process enters a new context. For example, in the Scrum methods, a sprint backlog is crafted when a new sprint is started. During the course of the sprint, elements are removed from the backlog immediately as they are classified as "done".	In many cases, the backlog would be better called "task list", as generally "logs" do not exist until something has happened. A task list is a very traditional and common item in projects.
Backlog item	See: "backlog".	

Term	Definition	Reflection from a traditional point of view
Burndown Chart	A visual representation of the progress of development, usually used inside an increment. It shows the number of hours remaining for the completion of an increment and is used to monitor progress and to estimate whether the planned tasks can be completed. It is usually represented in chart form that shows a downward trend of work left to do until "burning down" to zero. The chart is usually drawn from a task management system (even a digital spreadsheet) containing status for all current work items, the size of the items and estimated development velocity. It is the opposite of "burnup chart". See also http://en.wikipedia.org/wiki/Burn_down_chart .	The burndown chart is sometimes used in non-agile development to show an estimate of when implementation would be done. It would be very inaccurate, but still no more inaccurate than just about any other estimation method.
Burnup chart	A visual representation of the progress of development, usually used inside an increment. It shows the number of tasks completed in an increment. It is the opposite of "burndown chart".	Similar techniques are used in non-agile development to visualise progress, but they measure other things: requirements (which would be equivalent to stories) or work effort (which would correspond with story points). The progress is always compared with plans.
Business	In the agile context, business often refers to any function or activity in the organisation that isn't part of the development team or related to verification or validation.	Business is usually a more restricted term referring to the "core" business processes. Yet, any process to aid with software is often called a "business process". Therefore, the use of the term may cause confusion.
Business value	A value from the viewpoint of business, expressed by the business representatives, such as a product owner or any business stakeholder. Business value is sometimes defined as something that someone in the organisation is willing to pay for. Stakeholder and customers define value. It is important that the software developers do not make decisions of the value of any proposed product feature.	In traditional development anything that has been accepted as requirements, is seen to produce value. Prioritisation of requirements shows the magnitude of value. The amount of value is a vague concept, when a requirement is mandatory, such as a safety requirement.
C		
Certified ScrumMaster	An individual is certified when he or she has attended, participated in, and completed a Certified ScrumMaster course. Note that working as a ScrumMaster does not require certification. Courses are provided by training providers accredited by the Scrum Alliance. See: http://www.scrumalliance.org/pages/certified_scrummaster.	
Collocated teams	Project teams are working in the same location (the same site). It is the opposite of "distributed teams".	(No difference.)

Term	Definition	Reflection from a traditional point of view
Continuous integration	A low-level software integration process where a new build is generated as soon as a new or updated source code module is uploaded to the system, or at least many times a day. The build server monitors configuration management system or version control system and starts a new build automatically. See also http://en.wikipedia.org/wiki/Continuous_integration	In traditional development, integration is never done in "big bang manner", but usually done periodically or in a more flexible manner. So, it is not continuous, but be automated for example to start at given hour.
Cross-functional team	A team which is comprised of members from various company functions, such as software development, testing, marketing, hardware engineering etc. The purpose is to collect all disciplines and skill that are required to complete a project from start to finish into one team. The team should ideally have the authority to make most decisions regarding the project.	Representatives of company functions do not usually work as a team, but participate in processes as stakeholders and within planned actions. The team's authority is defined in the project plan and requirement specification. Anything that contradicts those, need negotiations outside the project team.
D		
Daily Scrum	In the Scrum method, a short (often 15 minutes) daily meeting where the team members share what they did the previous day and what they aim to do that day. The meeting offers a place to support others and to synchronise work in a small scale. Distributed teams should be included in the meeting by some method, such as teleconferencing; the time of the meeting should in that case be chosen to be suitable for all teams. See also: "Scrum".	This corresponds with a project team meeting, which are often held weekly. Traditional project teams will have many formal meetings on many issues and the total count of meetings may be higher than in agile development.
Development story	A story intended to develop code or application technology and not to directly implement a requirement or user story. See also: "story"	In non-agile development, this would simply be called more exactly "development task", which would not confuse the nature of the task.
Distributed teams	Project teams are working in different locations, at a different site, perhaps across the globe. It is the opposite of "distributed teams".	(No difference)
Distributed development	See "distributed teams".	(No difference)

Term	Definition	Reflection from a traditional point of view
Done	A product feature (or a "story") that has received all necessary design, implementation, testing and documentation tasks in the current increment's context; excluding any tasks reserved for a separate release process. Sometimes there are various levels on "done": 1) Done: implemented and runs on developer's workstation, 2) Done: technically verified by testing etc., 3) Done: Validated to be of releasable quality. So, when using the term, one must first find out the term's meaning and use in the current context. See also: "release process".	The concept of readiness would refer to project phases and in their viewpoint, a requirement would be similarly "ready" when the "requirement is accepted", when the designs linked to it are "reviewed", when the implementations of the designs are done and unit tested and again (up to the upward slope of V-model), the development of the requirement is ready when it has been verified and validated. So, mostly the control flow and viewpoint are different, but by nature, all software elements need to go through a similar process.
E		
Epic	A group of related user stories or a very large user story that needs to be split into individual user stories. May be a non-analysed set of ideas that need working out before assigning to be developed. See also: "user story", "theme".	This would be a result of user studies or the business needs, showing a wide range of activities that the software should support, which need to be analysed and transformed into requirements to enable their development.
Estimation	Estimation of effort or size of the items on a project backlog or an increment's / sprint's backlog. See also: "story point".	Estimation is a traditional task in any development process.
eXtreme Programming	A traditional agile method. Often referred to just as XP. This is the method that made agile development familiar and most visibly introduced many agile practices to companies. See: http://en.wikipedia.org/wiki/Extreme_Programming	
F		
Feature driven development	A development style where teams develop program features from start to finish, instead of developing components. The style leads to technically good implementations, but is seen not to necessarily produce a good architecture or a consistent user interface style. See: http://en.wikipedia.org/wiki/Feature_Driven_Development	Can as well be used in plan-driven development.
Five Ss (5S)	A Lean concept of how items on the workplace are stored. In an expanded version there are 8 Ss: Sorting, Straightening, Sweeping or shining, Standardising, Sustaining the discipline, Safety, Security and Satisfaction. This is a discipline used also in software development to "housekeep" processes, tools, work items, configuration etc. so that they are always in top shape. See: http://en.wikipedia.org/wiki/5S_(methodology)	This is a process independent concept and usually more emphasised in plan-driven processes where there may be more instructions on these issues (like how to arrange a project's data; what the naming conventions are, etc.). This is a practical matter of operational quality and quality management.

Term	Definition	Reflection from a traditional point of view
Five whys (5 whys)	A Lean culture technique where any root causes or hidden factors are identified by asking "why" until the real issue is revealed. The technique is used in extracting requirements (the real needs of a customer, behind a request) and in problem solving (the root causes of failures). See: http://en.wikipedia.org/wiki/5_Whys	(No difference.) Mostly known as a product requirement extraction technique.
Flow	A concept for how increased value or work products move from one task or phase to the other. The flow is one key Lean principle and should be as smooth as possible. See: "Lean".	
H		
Hybrid process	A development process that uses both agile and non-agile practices, or generally speaking, a process that combines elements from various paradigms in a unified process.	Most processes are somewhat hybrids as they supplement basic processes with additional or customised task that are necessary in a domain. Often that customisation is company specific. But all previous generic processes were hybrids that combined the best parts of previous processes.
I		
Impediment	An identified obstacle that could hinder or delay development. Can be divided into internal (for the team to handle) or external (for, for example, the product manager to handle). A list of those is sometimes maintained, called an impediment backlog.	This is related to a list of project risks in traditional project work and product development, where any possible obstacles are identified analysed and assigned for a person to handle; more often than not to the project manager.
Increment	A development cycle in an agile process which starts by planning the contents of it – what requirements, features, user stories and such should be developed during the increment. It continues with development work and ends in the assessment of the work done. An agile project consists of many increments. An increment is also called "iteration", but iteration can also refer to smaller scale iteration during the designing of, for example, a module or a feature. In Scrum, increments are called "sprints".	The concept of increment is used in many ways: releases, builds, new versions. Practically all development processes, no matter how plan-driven, have some incremental elements in them.
Inspecting and adapting	A practice where teams observe their success and identify changes in their context and goals and adapt their actions accordingly.	In traditional development, this happens when requirements change, when project environment or organisation changes and periodically, when a long project assesses its performance or anytime there are some problems.

Term	Definition	Reflection from a traditional point of view
Iteration	A repeated work flow that produces a changed (improved) work product. In the process context, see: "increment"	(No difference)
J		
Just-in-time design	A design that is waited until the designer fully knows what the design should be like, and only then creates the design. See also: "last responsible moment"	In plan-driven development the expected time for the design decisions is based on plans and should not be adjusted by personal decisions.
K		
Kaizen	Continuous, incremental improvement. The term is mostly used in the context of Lean. See: http://en.wikipedia.org/wiki/Kaizen	Continuous improvement is a feature of any organisation, based on traditional quality management principles and the requirements of quality management system standards (such as the ISO 9000 series).
Kanban	A lean methodology that includes a visual system for managing work, a minimised amount of work in process and a pull system to guide work items' flow through the system. See: http://en.wikipedia.org/wiki/Kanban	(Not usually used.)
L		
Last responsible moment	The last moment when a decision needs to be made. If it is made any later, it will delay the process. By deferring decisions, more information can be gathered to support them. It should be noted that other project participants may have a different idea of that time and may think that an earlier moment would give them more possibilities to utilise the decision and to adapt to it.	In plan-driven development the expected time for decisions is based on project plans and should not be adjusted by personal decisions. There is often a preference of early decisions that give teams and individuals the maximum time to plan their tasks accordingly.
Lean	An integrated set of values, principles and practices used in an organisation's activities. Originates in Japan from Toyota's car manufacturing practices and has been formulated as Lean software development and also used in many other areas. Key ideas include: customer orientation, collaboration, professionalism, flow, not making waste, pull-based workflows, continuous improvement and others. As it is an integrated set of ideas, no single idea should be seen as the "most essential" element of it – not "flow", nor "avoiding waste" or anything else. See: " Lean software development" See: http://en.wikipedia.org/wiki/Lean_manufacturing, http://en.wikipedia.org/wiki/The_Toyota_Way.	Lean is really not that agile and its principles are very much at home in any plan-driven activity.

Term	Definition	Reflection from a traditional point of view
Lean software development	An approach to software development based on the ideas of Lean production. Used in context of agile software development to complement it. Based on the ideas of "Lean manufacturing". See:" Lean" See: http://en.wikipedia.org/wiki/Lean_software_development, http://en.wikipedia.org/wiki/Lean_manufacturing	Lean is really not that agile and its principles are very much at home in plan-driven software development. Lean's goal of preventing errors and a thorough causal analysis of problems, respect for specialisation and a holistic view of organisation are some ideals of systematic engineering – also in safety-critical domains.
M		
Muda	A Japanese term for waste, used in the context of Lean. Any work that work product that doesn't contribute to the customers' benefit (value). Any unnecessary action; any action repeated unnecessarily; defects are also muda. See also: "Lean" See: http://en.wikipedia.org/wiki/Muda_(Japanese_term)	Simply: unnecessary work, duplicate work, errors or waiting. There is no need for a generic Japanese term. Everyone understands without another, foreign term, that defects or unnecessary waiting are not desired and should be gotten rid of.
P		
Pair programming	A working style where two developers work together on a single workstation. Code is developed in collaboration; one person at a time typing it in. See: http://en.wikipedia.org/wiki/Pair_programming	(Rarely used.)
Pig	Slang used in the Scrum method. The term refers to a person responsible for doing a task in the current sprint. The term is not insulting, but refers to a role in the project. See: http://en.wikipedia.org/wiki/The_Chicken_and_the_Pig	
Planning poker	A technique for estimating the effort or "size" of development tasks. It is based on teamwork and consensus. Can be used in many agile methods. It is based on a form of Delphi method. See: http://en.wikipedia.org/wiki/Planning_poker See also: http://en.wikipedia.org/wiki/Delphi_method	There are a large number of estimation techniques used, including techniques where the team estimate the effort required for the tasks by discussing the tasks and looking for consensus. Terms related to "games" (be they card or ball games) are not used in the method names. Playing games is not usually seen appropriate in any critical development.
Product champion	See: "product owner".	See: "product owner".
Product owner	A product owner "owns" the product under development, meaning that she/he has a vision for it, holds the responsibility for it towards the organization's management and the authority to make decisions about it. She/he decides during the development on the project's features and is responsible for deciding their priority. The role is closely related to a traditional product manager role. The role and the term are used in many agile software development methods.	A product manager is a traditional position in an organisation, usually positioned in a business unit. All the product owner's tasks are also a business product manager's tasks. There may also be a technical product manager who is responsible for the product technologies.

Term	Definition	Reflection from a traditional point of view
Pull downstream	A process control mechanism where no work is done unless the next process phase (the "customer" for the current phase) expects it. In a more general meaning, work isn't done, unless someone has demonstrated a need for the outputs of that work.	At the requirements level, this is a principle of good development: any activity requires a decided requirement, which means that someone expects it to be implemented and delivered.
R		
Refactoring	The practice of continuously improving a work product, usually software code to improve its quality. Often the aim is to keep the behaviour of the code similar and change the structure and other internal elements. Tests are used to verify that during and after the changes the behaviour was not changed. Considered to be an important part of test-driven development. See: http://en.wikipedia.org/wiki/Code_refactoring, http://en.wikipedia.org/wiki/Database_refactoring, http://en.wikipedia.org/wiki/Test-driven_development	In traditional development, the rewriting of code would be "rework" and considered harmful. The goal is to write sufficiently good code from the beginning. Sometimes the code is reviewed and actually frozen until a defect causes a need to repair it. Improving plans and designs, on the other hand is always seen as positive, if it is done during a planning or designing phase of the project. After that, it should be avoided.
Release plan	A plan of a project's releases. A rough plan of suggested releases (a roadmap) and / or a plan for a single release, its processes, tasks, resources etc. A generic term used in many development methods, including non-agile ones.	In traditional development, exact requirements or features can be mapped to the planned releases, so the planning is stricter.
Release process	The process of turning a development team's product into one that can be published and made available to customers. Includes validation that all necessary requirements have been met, all required certifications have been acquired, starting any support functions etc.	(No difference.)
Retrospective	A meeting held at the end of a process increment (in the Scrum, after every sprint) where the project's success is reviewed: what went well, what were the problems, how can the problems be solved in the future and what can be improved. Retroperspectives can form an important element in continuous improvement of the organisation. Retroperspectives are also called Lessons Learned meetings.	The process is the same but meetings are held not so often; perhaps when a project end (in which case the findings can only help next projects) or for example at the end of project phases.
Roadmap	See: "release plan".	In traditional development, a roadmap usually refers to a product's planned technological advancement though various projects and life phases; perhaps projecting many years to the future.
S		
Scrum	One agile software development project management method. See: http://en.wikipedia.org/wiki/Scrum_(development) , http://www.scrumalliance.org/	

Term	Definition	Reflection from a traditional point of view
Scrum Master	A role in the Scrum method. The Scrum master acts as a facilitator for the team, oversees the team's progress, facilitates daily meetings and removes obstacles. The role is sometimes said to be a "servant leader". Note that the term is just as often written as ScrumMaster. See http://en.wikipedia.org/wiki/Scrum_(development)#Roles See also: "Certified ScrumMaster"	There is a resemblance to a good, professional, personnel-oriented, project manager or team leader. There are good and bad management and leadership practices, and Scrum's differentiation of terms could be seen to make it clear that mechanistic, bad management is not expected any more in software development. Instead, facilitation and leading of group dynamics are needed more.
Scrum meeting	Any meeting in a Scrum method, including a planning meeting, a review meeting, a retroperspective and a daily Scrum.	Any project meeting.
Self-directing team	A team that decides by itself its future actions, for example what development goals they should tackle first. See also: "self-organising team"	Used rarely outside R&D type development where a team is given a freedom to seek new, radical solutions.
Self-organising team	A team that decides by itself its members' roles and division of tasks. Based on the idea that the ones closest to a task know best how it should be done and therefore can best divide the work themselves. See also: "self-directing team"	No difference. Test team can sometimes work quite in this manner.
Sprint	See "increment"	
Sprint backlog	See "backlog"	
Sprint planning meeting	A phase in the Scrum methods where the contents of the next sprint are planned.	Project phase planning meeting. A project may have rough previously-made plans for all its phases, but still, each new phase requires planning and meetings when it is started.
Sprint review	A meeting in the Scrum method at the end of a sprint where the teams and stakeholders assess the completed tasks based on a demonstration.	Project phase review meeting where the completed tasks of the phase are assessed and perhaps accepted so that the project can move to the next phase.
Stand-up meeting	A meeting where participants do not sit down. Most famously used as a style of daily meetings in the Scrum method, the Daily Scrums.	No conceptual difference, but meetings are very rarely held standing up.

Term	Definition	Reflection from a traditional point of view
Spike	An investigation of feasibility of a product feature / technology based on an implementation such as a rough prototype or mock-up. Usually done in a timeboxed fashion, often rapidly between increments. It enables the estimation of effort of implementing a requirement / story. Term is used only in agile software development.	The task of creating a mock-up, a prototype or a technology demonstration that verifies feasibility and makes estimation possible. A feasibility study would usually refer to a task before a project where it is assessed whether the whole project is feasible.
Stakeholder	An external (from the viewpoint of an agile team) organisation or a person representing the organisation who has an interest in the product.	(No difference.)
Story	A work item, most often a product requirement expressed from a user's point or view, but also any other project task, including a testing task. See: "user story", "development story", "analysis story".	Generally, this is a task and a task is called a task. A user story is an exception (See: "user story")
Story checklist	A list of elements that need to be included in a story's description for it to be complete.	Task description template.
Story point	A context-dependent measure of effort of implementing software that enables a user story. Used in estimating effort and measuring development velocity. Sometimes numbers (in particular Fibonacci numbers) are used in assigning a story a number of story point and thus giving a relative size to a story (1, 2, 3, 5, 8 where 1 could mean low complexity and 8 the task to be very complex). See also: "user story", "velocity"	Effort estimate unit for a task (hours, function points, lines of code etc.).
Story time	An informal part of the Scrum method. Meetings where the backlog items are discussed, refined and prioritised and the backlog otherwise maintained. These meetings are held during the sprints.	Just a regular team meeting, in which perhaps the product manager participates.
T		
Task board	A physical board or a view in an information system showing the state of tasks in a current increment. The tasks may be written on sticky notes ("yellow notes") and place on a wall board in columns labelled, for example, "to do", "in progress" and "done". As work progresses, each task (its note) is moved from one column to the next. Thus, a task board is an efficient and practical tool for tracking a team's progress. The same information can obviously be expressed in a task management system, allowing views to the progress to the outside of the room – to the stakeholders, to other teams, to other sites.	This would correspond with a Gantt chart on a project room wall where the progress of tasks is shown and updated by the project manager. Of course, the similar view usually needs to be provided in an information system to share the progress information with other teams and stakeholders. The Gantt chart would include progress information from other teams, too.
Teamlet	A temporary sub-team of a (development) team that forms to work on a single work item.	Conceptually no difference, but the actual term is rarely used.

Term	Definition	Reflection from a traditional point of view
Test-driven development (TDD)	An approach to development, unit / module tests are written for functional code before the actual code. First the tests fail, but when the code is written, tests start to pass, one at a time. Once the tests pass and remain in place of the personal build environment (usually the IDE), code can be improved (refactored) as the tests verify that the external behaviour and thus the functionality stays the same. This is called a "safety net" for errors and regression. Thus, the process is evolutionary, quality-conscious and safe. See: http://en.wikipedia.org/wiki/Test-driven_development See also: "refactoring".	No difference. While mostly talked about in the agile context, this would be a natural way to start implementing a module. Still, in traditional development the unit testing practices are not as critical due to less changes and to an emphasis of more up-front architecture design.
Technical debt	According to Wikipedia: Technical debt and design debt are synonymous, neologistic metaphors referring to the eventual consequences of slapdash software architecture and hasty software development. Code debt refers to technical debt within a codebase. See: http://en.wikipedia.org/wiki/Technical_debt	(The term is not necessarily an agile one, but seems to be more often used in the agile context.)
Theme	A top-level objective that may span projects and products. A strategic tool for guiding development and align activities. An epic can be considered a theme. See: "epic".	See: "epic".
Thumb vote	A simple voting technique in a meeting where thumb up or down represents an opinion on an issue.	It is a normal meeting practice and human behaviour to ask participants' opinion. This is just one way to express it. (In Finland, people would usually raise their hand in a similar situation.)
Timeboxing	Development work is constrained by giving it a non-negotiable time limit (e.g. the duration of an increment). The amount of work is adjusted to that constraint during the planning and during the development by dropping tasks if it is seen that they cannot be completed during the time allocated. Increments of agile processes are practically Examples include iterations, spikes and stand-up meetings. See: http://en.wikipedia.org/wiki/Timeboxing	A project's deadline could be considered a time-box, if it is decided before the project's contents. The same applies to any situation where meeting a deadline is more important than anything else.
U		
User acceptance testing	See: "acceptance testing".	See: "acceptance testing".
V		
Value stream	Set of actions from the beginning of the project (or a request from a customer) to the delivery and deployment of the product.	(No difference.)

Term	Definition	Reflection from a traditional point of view
User story	A story of user behaviour that presents something that the new version (increment) of software should enable. Often expressed in first-person: "As a user, I want to be able to..." Usually concise descriptions that fit on a piece of paper (a large sticky note / yellow note). The effort of implementing a user story is measured in "story points". Thus, it can be estimated how many user stories the team can complete during an increment (the development velocity). See: http://en.wikipedia.org/wiki/User_story See also: "story", "story point", "velocity"	In traditional development, these are represented in various ways, for example, a requirement, a use case, a business process description. In user-centred development the concept of a user / use scenario is used in almost the same way.
Velocity	A measure of development speed, often measured in how many story points or user stories a team can implement during an increment. But the actual metrics used can vary, depending on development context and culture.	In traditional development, other metrics are used. The application of velocity metric is different. In agile development, the measured velocity can be used in estimating the next increment, but in traditional development it mostly aids in the planning of the next project.
Vision statement	A high level document that describes the product vision: why it should be developed, who it is for, what the concept is, what value it provides, how it differentiates from other products etc. A statement like this gives development motivation and direction.	The description of vision is common practice also in non-agile software development, sometimes by that name, sometimes in the context of the system's concept definition.
Voice of the customer	A mostly Lean-originating term meaning that the end customer's and user's opinions need to be captured in their original form (although transferred to other forms during the process). User stories are a technique to retain the voice of the customer during the development process. Note that while the voice of the customer is important, in the development process it is important to capture the real needs of the customer and not just the expressed wishes (this is the purpose of the technique of asking "why" until a read need emerges – the five whys). See: http://en.wikipedia.org/wiki/Voice_of_the_customer See also: "user story", "five whys"	(No difference.) Historical note: The QFD method was known and still remembered for its use of the voice of the customer to assess and trace product requirements and features. See: http://en.wikipedia.org/wiki/Quality_function_deployment
W		
Waste	See: "muda".	See: "muda"
Work in progress	Work which has been started but not yet completed.	(No difference.)
X		
XP	See: "eXtreme Programming"	