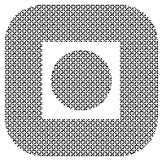
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatens navn:	Liv Ryssdal Thorsen
Fag:	SIF80/H02
Oppgavens tittel (norsk):	Extreme Programming i sikkerhetskritiske systemer
Oppgavens tittel (engelsk):	**Extreme Programming in safety-related systems**

Oppgavens tekst:

Safety-related systems place restrictions on the development process – for instance IEC 61508. Can these requirements be satisfied if developing according to XP?

Oppgaven gitt:	20.01.2002
Besvarelsen leveres innen:	17.06.2002
Besvarelsen levert:	17.06.2002
Utført ved:	NTNU
Veileder:	Professor Tor Stålhane

Trondheim, 17.06.2002

Professor Tor Stålhane
Faglærer

Extreme Programming
in
safety-related systems

Preface

This report is the result of the diploma work at the Department of Computer and Information Science, NTNU spring 2002. The purpose of the diploma is to evaluate whether Extreme Programming is suitable for development of safety-related systems according to the IEC 61508 standard.

I will especially thank my supervisor Professor Tor Stålhane, Norwegian University of Science and Technology, Dept. of Computer and Information Science for all contributions and guidance. Thanks to all participating persons; Reseach Scientist Odd Nordland, SINTEF Telecom and Informatics, Professor Tor Onshus, Norwegian University of Science and Technology, Dept. of Engineering Cybernetics, Professor Torbjørn Skramstad, Norwegian University of Science and Technology, Dept. of Computer and Information Science, for their time and valuable thought about different approaches discussed in this report. I am grateful for Tor Einar Kjørkleiv's review of the report.

Trondheim, June 17, 2002

Liv Ryssdal Thorsen

Abstract

This report discusses whether Extreme Programming (XP) can be employed when developing safety-related system if conformance to IEC 61508 shall be met. XP is a recent methodology for software system development. It focuses on short iterative development, high customer integration, extensive testing, code-centered development and documentation, refactoring, and pair programming. IEC 61508 on the other hand includes both hardware and software development of the system, and uses the standard V-shaped development model. We consider the following the most important contributions of this report. First, from our comparison of XP and IEC 61508 we are able to identify several confliction and non-confliction areas. Second, we propose how these methodologies can be brought closer by proposing a software safety model. Third, our suggestions include some conditions that can be a subject to further analysis.

The recommendations are based upon a literature research and discussions with domain experts. The proposal stated in this report includes adoption, either partly or totally, of ten out of twelve key practices in XP. Practices that are totally adopted are: the planning game, simple design, testing, pair programming, collective ownership, 40-hour week, on-site customer, and coding standard. None of these where in conflict with the requirements in IEC 61508. Refactoring, and continuous integration where only partly adopted. Refactoring because one has to be careful when modifying safety-related systems, and therefore care should be taken when one wants to change the code. Continuous integration depends of the availability of the hardware. If the hardware is developed in parallel with the software, it is not possible to integrate continuously.

The proposal includes a modified software safety lifecycle, where all the phases presented in the software safety lifecycle in IEC 61508 are represented in the new model. This model explicitly specifies iterations, and makes it possible to adopt changes to the software. The system is based on a stable architecture that should not be modified, but minor requirement changes are possible with use of refactorings.

In our opinion this proposal have several advantages. The iterative development adds flexibility to the system, since requirements can be added or changed if they fit the architecture. Automatic testing makes the verification of the changes in code easy. In addition the customer is part of the XP team and this makes it possible to give early feedback and guidance. The programmer can constantly ask the customer if something is unclear. Knowledge about the system is spread through the team due to collective code ownership and pair programming. We think that the proposal stated in this report can lead to faster development, better predictability, more job satisfaction, and achieve an appropriate level of safety with as low cost as possible.

Table of contents

Table of figures

Table of tables

1 Introduction

This chapter provides information about the post-graduate thesis, its scope, a brief overview of related work, and how the report is structured.

1.1 Motivation

The working title of the given assignment had the title "Extreme Programming in safety-related systems", and the following paragraph was given as a guideline:

> *Safety-related systems place restrictions on the development process – for instance IEC 61508. Can these requirements be satisfied if developing according to XP?*

Extreme programming, introduced in 1996, is a lightweight software development methodology. It departs significantly from traditionally development practices, including development of safety-related system. In contrast to the sequentially development of safety-related system, XP is oriented towards delivering incrementally growing software product. It gives preference to informal oral communication in development team over methods of written documentation. Industrial interest in the use of incremental development and XP is rapidly growing. This can be seen as a reaction to earlier, extensive and document driven development models.

When developing safety-related system it is crucial to be able to verify that the final system is adequately safe. Standards are used as means to ensure a sufficient level of safety. Development of safety-related software has strict requirements on how the development should be performed. Many standards are relevant for development of safety-related system. The international standard IEC 61508 was published in full in 2000, and are therefore one of the most recently published safety standard on the marked. The aim of IEC 61508 is to provide directions whereby safety-related systems can be implemented in such a way that an acceptable level of functional safety is achieved. It has the advantage over national standards of being a reference source across national boundaries and therefore a common basis for determining working practice. Since it is generic, it can be applied to different kind of safety-related systems in many industries. It has been conceived with a rapidly developing technology in mind and will therefore not easily be outdated. This is some of the reasons why it is already widely used in the industry. IEC 61508 has the opinion that the development process must be planned and controlled in order to achieve the best result. The development of safety-related systems follows conservative development methods. The safety community does not want to use new methods before they have been tested thoroughly. Despite the refusal of adoption of new methods we want to investigate the possibility of using XP. This methodology is a modern, lightweight development methodology, and many have seen its advantages. Use of IEC 61508 together with XP faces several challenges. In this report we look into these challenges and give some proposals on how to manage them. Industry are faced by high pressure to reduce costs coupled with shorter product life cycles, and a need for a quicker time to marked. We will investigate whether XP can help to accomplish these goals.

1.2 Scope

In this report we will only focus on the standard IEC 61508, due to its widespread use and applicability to many types of safety-related systems. It applies to safety-related systems when one or more of such systems incorporate electrical and/or electronic and/or programmable electronic (E/E/PE) devices. It is based on and applicable to all E/E/PE safety-related systems irrespective of the application. Another reason for its widespread use is its generic nature, it is a framework for developing domain-specific safety standards.

Safety can only be assured by considering all aspects of a system, including both hardware and software. It is a system issue. IEC61508 covers both the hardware part and the software part of the development. In this report we will only focus on the software aspect of the system development.

In addition to the development process, XP has opinions about other areas, such as facility strategies and working hours. We will only discuss XPs practices concerned with the development process, other areas are not with in the scope of this report.

1.3 Related work

Grenning wrote an article [17] about adaptation of XP in a company with a large formal software development process. The division, that started adopting XP, was developing safety-critical systems. Previously they used up-front requirements documents, up-front design, reviews, and approvals. They confirm the different view of documentation XP stated. XP acknowledges that writing documentation requires resources, and therefore less time is spent on developing the system. The division used to write extensive documentation of the system, but now they saw the possibility of changing that practice. Therefore they listed what they wanted from documentation:

- Enough documentation to define the product requirements, sustain technical reviews, and support the system's maintainers.
- Clean and understandable source code.
- Some form of interface documentation, due to the impedance mismatch between groups.

They saw the benefit of documenting what they had built, not what they anticipated to build. The significant designs were documented within an iteration and reviewed with the review team. On the XP team, dependencies between features were almost nonexistent. They built features in the order of customer priority, not internal software framework order dictated by a Bid Design Up Front (BDUP).

Other methods or techniques they found useful were:

- Use cases
- Clean and simple source code for maintenance purposes
- High-level document to navigate in the system
- Monthly design reviews
- Ask-for-forgiveness design process

The article does not say whether they developed systems according to a standard. Grenning saw many of XPs advantages and they tried to do as many of the XP practices as they could. They thought XP right out of the book would not work for them. Documentation and reviews were going to be the big roadblocks.

The development of safety-critical software has been widely addressed in literature. Several books and articles have also been written about XP. However, an explicit account of whether XP can be used in developing safety-related system is not available: a gap we are trying to bridge with this report.

1.4 Research agenda

The result from this thesis is drawn from literature research and discussion with safety experts who have experience with certification of safety-related system and knowledge of the requirements stated in IEC 61508. The literature research was comprised of collection and structuring of relevant documentation of XP, safety-related system, IEC 61508, and other relevant areas.

This report is mainly organized as a comparison between XP and IEC 61508. First a deep literature research on the practices found in XP was carried out. Then we proceeded with a similar study for IEC 61508. Important methods and requirements in the IEC 61508 standard are discussed. Based on these findings we investigated whether the practices XP represents are sufficient to fulfill these requirements. Is this requirement in contradiction with XP? How could this requirement be confirmed when developing according to XP? Which approaches have to be done to satisfy XP and the IEC 61508 requirements? What practices are not in contradiction?

1.5 Report structure

This report consists of seven chapters.

Chapter 1 Introduction: provides information about the thesis, its scope, a brief overview of related work, research agenda, and how the report is structured.

Chapter 2 Extreme Programming: contains an overview of the Extreme Programming (XP) methodology. We have split the practices into four areas: planning, designing, coding and testing. Refactoring, which is an important part of XP, is discussed next. Finally, the role for people involved in an XP team is described.

Chapter 3 IEC 61508: presents the international standard IEC 61508. First we give a brief introduction to safety-related system. The introduction places IEC 61508 in a wider perspective. Next an overview of IEC 61508 is given, and important parts of the standard are discussed in more detail. This concerns the safety lifecycle, hazard and risk analysis, safety integrity level, and ALARP. Finally we select three key areas, and investigate what has to be fulfilled in order to claim compliance with the standard.

Chapter 4 Comparison of XP and IEC 61508: a comparison of XPs practices and IEC 61508s requirements to software development is presented. First a high-level comparison

is introduced, then the individual steps of planning, design, coding, validation and verification, management, and documentation are further analyzed.

Chapter 5 Compatibility of XP and IEC 61508: contains approximations between XP and IEC 61508. The fundamental differences discovered in Chapter 4 are further discussed. A proposal for a new software safety lifecycle is presented. To conclude, the adoption status of the XP practices that we make use of in the proposal are summarized in a table.

Chapter 6 Conclusion: brings all the investigated pieces together. We summarise the proposal, and the most important arguments why it will improve the development of safety-related system.

Chapter 7 Further work: provides recommendations for further work based on the findings in this report.

2 Extreme Programming

Extreme programming (XP) is a lightweight software development methodology developed by Kent Beck, and others. Kent Beck wrote the first book **[1]** about XP in 2000, therefore it is one of the most recently published methodologies. The methodology comes without much overhead so it can be applied easily. It attracts attention from many software development teams and its popularity is growing fast. The XP practices have been selected to form complementary sets in which one of them supports another **[1]**.

Despite the fact that it is a newly published methodology there is little new in XP. Its practices have been around for years. The difference is that XP does not restrict these practices to distinct "phases" of a project; they are done all the time. The "extreme" in XP comes from two things **[10]**:

1. XP takes proven industry best practices to extreme levels. For example, designing, code review, testing and refactoring are done continuously, rather than at dedicated phases of the software process only.
2. XP combines those practices in a way that produces something greater than the sum of the parts.

2.1 Practices

XP is oriented towards delivering incrementally growing software products. It gives strong preference to informal oral communication in development teams over written documentation of design. The source code plays an important role in the development process: code is documented via test code, the tests themselves are code rather than input data, and continually code refactoring can make the overall design simple. Direct contact with the customer representative leads to introducing changes to the project at early stages. The best strategy is to embrace changes, and perform a quality work. If the programmers do a good job, they will enjoy working and work well.

Below follows a list of rules and practices of XP. As in **[37]** and **[59]**, the practices are split into four areas: Planning, Designing, Coding and Testing.

2.1.1 Planning

P1 *The planning game is used to create project plans.* The main idea behind the planning game is to make a rough plan quickly and refine it as things become clear. A customer representative is making business decisions (choosing the project scope, development task priorities, composition of releases, dates of releases, etc.) and developers are making technical decisions (evaluating risk factors, estimating the effort, system design, process, etc.) **[1]**.

P2 *The project team is traveling light.* The only artifacts the team is writing and using are test cases and code.

P3 *User stories are written.* They serve the same purpose as use cases. Customer writes them and they are used instead of requirement documents. The developers estimate how long each user story will take to implement.

P4 *Release planning creates the schedule.* The goal of the release planning game is to define the set of features required for the next release.

P5 *Make frequent small releases.* Release the system as soon as it makes sense (after one to two months). Having a running system with reduced, but incrementally growing functionality, makes it possible for the client to give quick feedback. Such an approach minimizes costs of radical changes in the project because new needs and requirements can be taken into account in one or more of the nearest increments.

P6 *The Project Velocity is measured as a metric.* This shows how fast work is getting done. Effort is estimate in Ideal Engineering Time, which assumes no interruption, no meetings etc. The Project Velocity describes the amount of Ideal Engineering Time per month **[1]**.

P7 *Iteration planning starts each iteration.* A plan of the programming task for that iteration is made.

P8 *Move people around.* This refers to moving people around the code base in combination with pair programming. Developers exchange programming partners frequently.

P9 *A stand-up meeting starts each day*. The purpose of this meeting is communication among the entire team e.g. quick review of status, requests for help, problems encountered, and discoveries made. Naturally, customers and managers are invited, and should attend.

P10 *Fix XP when it breaks.* The team can change what does not work. It is not the manager's job to dictate what to change and how, but to point out the need for change. The team should come up with one or more experiments to run.

P11 *Accepted responsibility*. There is no top-down planning in XP, responsibility can only be accepted, not given.

P12 *Planning for priorities*. The highest business priorities are implemented first.

2.1.2 Designing

D1 *Small initial investment*.

D2 *Simplicity.* Make the solution as simple as possible. According to Kent Beck the simplest design that could possible work; runs all the tests, contains no duplicate code, states the programmers' intent for all code clearly, and contains the fewest possible classes and methods. This rule can be summarized as, "say everything once and only once." The more complicated things are, the harder it is to keep them under control.

D3 *Choose a system metaphor.* Development is based on a simple story of how the system works. The metaphor is a communication means between customer representatives, managers, and developers. It gives the team a consistent picture they can use to describe the way the existing system works, where new parts fit, and what form they should take.

D4 *Use CRC cards for design sessions.* CRC cards (Class, Responsibilities, and Collaboration) allow the entire project team to contribute to the design.

D5 *Create spike solutions to reduce risk.* A spike solution is a small, informal experiment with an idea on how to solve a problem.

D6 *No functionality is added early.* Extra functionality will slow the team down and waste resources.

D7 *Refractor whenever and wherever possible.* Refractoring is changing system's implementation without changing its behavior. The aim is to increase readability, flexibility, understandability etc.

2.1.3 Coding

C1 *On-site customer.* The customer representative is always available to the developer. Whenever the programmer is uncertain concerning a user story interpretation, the customer can explain. He also participates in the planning game and works on acceptance tests.

C2 *Code must be written to agreed standards.* A coding standard is necessary because code written by one pair of programmers will be read and modified by other pairs.

C3 *Code the unit test first.* This means that a programmer starts coding after the test cases are written. This helps to understand what can be expected from a unit and removes "implementation bias" when testing a unit.

C4 *All production code is pair programmed.* Working in pairs on a single computer provides instant code review and is reported by programmers as more enjoyable than individual work **[60]**.

C5 *Only one pair integrates code at a time.*

C6 *Integrate often.* Kent Beck calls this practice "continuous integration" and suggests integration "*after a few hours (certainly no more than a day)*" **[1]**.

C7 *Use collective code ownership.* Any person on the team can change any piece of code if necessary. Everybody owns the code, meaning everybody is responsible for it: "You break it, you fix it." The unit tests must run all the time. If someone break something, it is his responsibility to fix it.

C8 *Leave optimization till last.* Make the system work, make it right, then make it fast.

C9 *No overtime.* Kent Beck says that he wants to be "fresh and eager every morning, and tired and satisfied every night." Therefore he suggests a 40-hour work week.

2.1.4 Testing

T1 *All code must have unit tests.* "Unit tests are written by the developers, using the same programming language as is used to build the system itself. Tests are small, take a white box view of the code, include a check on the correctness of the results obtained, comparing actual results with expected ones. **[11]**" XP prescribes to test *everything that could possibly break* **[30]**.

T2 *All code must pass all unit tests before it can be released.* Unit tests give developers confidence that their code works.

T3 *When a bug is found, a new test must be created.* This allows discovering early if the bug reappears in the future.

T4 *Write test before refractoring.*

T5 *Acceptance tests are run often and the score is published.* Those tests are written by the customer representative and they show the status of the project from the customer point of view.
T6 *Other tests*. Other tests besides unit test and functional test can be used if needed.

XP is based on four values that reinforce each other, they are practiced constantly through the development process. They may be summarized this way:

1. **Communication.** XP makes it next to impossible not to communicate by employing practices that can not be done without it. For examples unit testing, pair programming, and estimating.
2. **Simplicity.** XP proposes to do the simplest thing that could possible work. Beck says that *XP is making a bet that it is better to do a simple ting now and pay tomorrow to change it if necessary, than to do something more complicated now that might not be used anyway* **[1]**.
3. **Feedback.** Feedback early and often gives opportunity to "steer" the efforts.
4. **Courage.** Beck's opinion is: *if you aren't moving at top speed, you'll fail* **[1]**. When developers have courage they have the guts to make action when it counts, such as when code need to be thrown away or changes have to be made late in the game.

Four variables will be controlled in the project – cost, time, quality, and scope. Kent Beck says, "*of these, scope provides the most valuable form of control*" **[1]**. External forces as customers and manager get to pick the values of three of the variables. The development team gets to pick the value of the fourth variable. For example, the external force can require to get a system following the quality standard ISO 9000 within two month at a price of $ 900 000. The resultant value is scope, and therefore the programmers can decide how many system requirements they can implement.

Development in XP must be regulated by dynamic discipline. Architecture is validated by small, throwaway prototypes. Requirements are detailed by regular consultation between developers and customers. Quality evolves through automated testing of features and the units that comprise them.

2.1.5 Release

There are three phases in the Planning Game. A release starts with an exploration phase, in which business and development discuss what the system should do. The customer writes stories for feature requests, and the developers estimate how long each story will take to implement. If a story is too complex to estimate, it is split into smaller stories. In the subsequent commitment phase the stories are ordered by business priority, sorted by value, and sorted by risk. Then velocity is set, and scope chosen. These stories then get implemented in the longest phase (steering phase), which consists of a series of small iterations (see Figure 1). Iterations should not take more time than one month (one to three weeks is best). The plan is updated based on what is learned **[1]**.

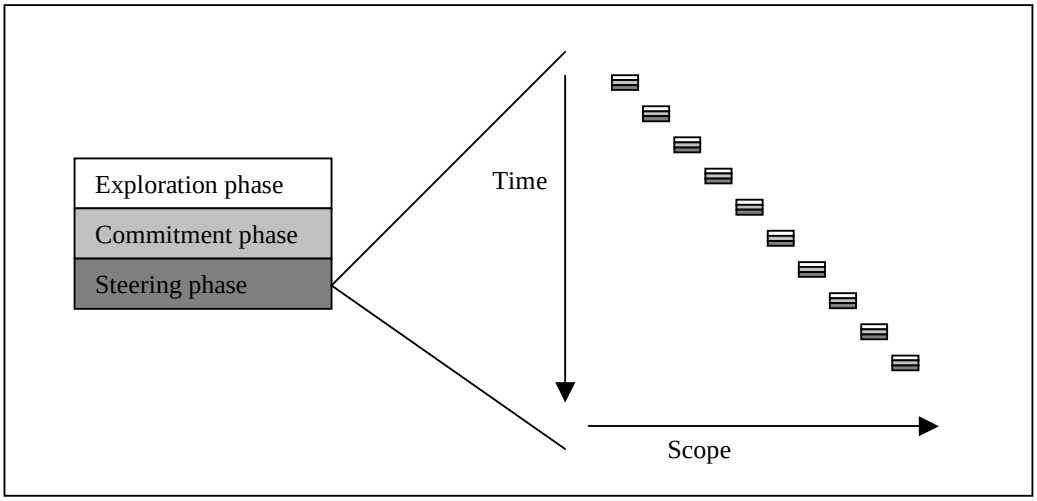


Figure 1 Release in Extreme Programming

2.2 Refactoring

Continuous integration, testing, and refactoring make evolutionary design efficient. Refactoring is an important component of XP. This subchapter contains a short introduction.

2.2.1 Introduction

Martin Fowler presented two definitions of refactoring, depending on context **[12]**:

Refactoring: a *change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*.

Refactoring: *to restructure software by applying a series of refactorings without changing its observable behavior.*

Refactoring is cleaning up code in an efficient and controlled manner. The purpose of refactoring is to make the software easier to understand and modify. Refactoring can also help the programmer to understand some code that needs to be modified. In addition, it can improve design, improve readability, and reduce the number of bugs. All this improves quality. Even though refactoring can lead to simpler designs it does not weaken flexibility. The most common time to refactor is when a new feature shall be added to some software. Refactoring can also help when a design does not let the programmer add a feature easily.

Most of the performed refactorings is low level, such as creating or deleting a class, variable, or function, or changing attributes of variables and functions, such as their

access permissions and functions, or moving variables and functions between classes. A smaller set of high-level refactorings are used for operations such as creating an abstract superclass, simplifying a class by means of subclassing and simplifying conditionals, or splitting off part of an existing class to create a new, reusable component class. The more complex refactorings are defined in terms of the low-level refactorings. Fowlers approach was motivated by concern for automated support and safety **[12]**.

2.2.2 Refactorings

Fowler has collected a catalog of refactorings, which range from simple modifications to more substantial changes comprising several different smaller refactorings. He grouped the refactorings after the purpose they serve. Table 1 shows a selections of refactorings for each group proposed by Fowler **[12]**.

Table 1 Examples of refactorings

Composing methods:	**Simplifying conditional expressions:**
Extract method	Decompose conditional
Inline method	Consolidate conditional expression
Replace temp with query	Consolidate duplicate conditional fragments
Introduce explaining variable	Remove control flag
Split temporary variable	Replace nested conditional with guard clauses
Moving features between objects:	
Move method	**Making method calls simpler:**
Move field	Rename method
Extract class	Add parameter
Inline class	Remove parameter
Hide delegate	Separate Query from modifier
	Parameterize method
Organizing data:	
Self encapsulate field	**Dealing with generalization:**
Replace data value with object	Pull up field
Change value to reference	Push down field
Change reference to value	Extract superclass
Replace array with object	Extract interface
	Replace inheritance with delegation
Big refactorings:	
Tease apart inheritance	
Convert procedural design to objects	
Separate domain from presentation	
Extract hierarchy	

2.2.3 Bad smells in code

Refactorings are applied when *bad smells* are detected in the code. Examples of *bad smells* **[12]**:

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change (changed in different ways for different reasons)
- Shotgun surgery (a lot of small changes to a lot of different classes)
- Feature envy (a method seems more interested in a class other than the one it actually is in)
- Data clumps (Bunches of data that hang around together ought to be made into their own object.)
- Primitive obsession (Replace data value with object, replace type code with class, etc.)
- Switch statements (when you see a switch statement you should consider polymorphism.)
- Parallel inheritance hierarchies (when you have to make a subclass of one class, you also have to make a subclass of another)
- Lazy class
- Speculative generality
- Temporary field (an object which an instance variable is set only in certain circumstances)
- Message chains
- Middle man (delegating to other class)
- Inappropriate intimacy
- Alternative classes with different interfaces (methods that do the same thing but have different signatures for what they do.)
- Incomplete library class
- Data class
- Refused bequest (subclass that does not need inherit method)
- Comments (often used as deodorant. When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.)

2.2.4 Problems

Something that is disturbing about refactoring is that many of the refactorings do change an interface. There is no problem changing a method name if you have access to all the code that invoke that method, but it is a problem if the interface is being used by code that you cannot find and change. One has to retain both the old interface and the new one, at least until the users have had a chance to react to the change.

2.2.5 Conclusion

Refactoring can be an alternative to upfront design. One just code the first approach that comes into your head, get it working, and then refactor it into shape. Even though

extreme programmers use refactoring, they design first. They will try out several ideas with CRC cards or the like until they have a plausible first solution. Only after generating a plausible first shot they will code and then refactor. The point is that refactoring changes the role of upfront design.

2.3 Roles for people

XP requires five roles to make the team function – customer, programmer, manager, tracker, and a coach.

2.3.1 Customer

The customer is the person or group who represent the users. The customer is responsible for identifying the features (stories in XP) that the programmers must implement, providing detailed acceptance tests for those stories and assigning priority to them.

2.3.2 Programmer

The programmers have the primary role in XP. They implement the stories written by the customer, and pass any tests that the customer specifies. Besides programming they are also responsible for estimating how long it will take to implement the stories.

2.3.3 Management

There are three prominent roles in managing XP:

- *Manager* manages the team and fix problems. He face outside parties, and obtain resources.
- *Tracker* helps the team to know if they are on track for what they have promised to deliver. This is a part-time role.
- *Coach* helps the team to use and understand the XP approach. He mentors the team, and handles problems.

A team may organize these roles in the way they prefer, but each of these areas must be addressed.

2.4 Summary

Extreme programming is a lightweight methodology for small teams. It is suitable for vague and rapidly changing requirement. Table 2 summarizes the twelve key practices of XP.

Table 2 The twelve key practices of Extreme Programming [1]

Planning game	Quickly determine the scope of the next release. Development estimates user stories, and the customer prioritizes them.
Small release	A simple system is put into production quickly, and then new versions are released on a very short cycle.
Metaphor	All development is guided with a simple shared story of how the whole system works.

Simple design	The guiding design principle is to do the simplest thing that could possibly work. Extra complexity is removed as soon as it is discovered.
Testing	Unit tests and acceptance tests are run continuously
Refactoring	Continuously improve the design without changing the functionality.
Pair programming	Production code is developed by pairs of programmers.
Collective ownership	Developers can modify any piece of code.
Continuous integration	Integrate changes immediately instead of developing them in separate branches.
40-hour week	Programmers work 40 hours max, to keep them fresh and creative.
On site customer	A customer is on the team to discuss feature requests and domain concepts.
Coding standards	Ensure agreement on simple coding conventions.

All the practices should be implemented to get the maximum benefit. There are two things in addition to the practices that are important for success – verbal communication and human relations. “If members of the team do not care about each other and what they are doing, XP is doomed” **[1]**.

The last rule stated by Kent Beck is “By being part of an Extreme team, you sign up to follow the rules. But they are just rules. The team can change the rules at any time as long as they agree on how they will assess the effects of the change” **[2]**.

3 IEC 61508

In this chapter we give a brief overview of safety-related systems (subchapter 3.1), and a comprehensive overview of the standard IEC 61508 from The International Electrotechnical Commission (IEC). Within this chapter we are primarily interested in the international standard IEC 61508, although the standard is linked with safety-related systems. The safety-lifecycle presented in IEC 61508, hazard analysis, safety integrity level, ALARP, and how to achieve compliance with the standard are further discussed. Subchapter 3.3 is concerned with the safety-lifecycle presented in IEC 61508. It is used as a framework to structure IEC 61508s requirements. Hazard and risk analysis, presented in subchapter 3.4 are used to investigate the safety implications of a system. The former looks at the identification of situations that could endanger human life or the environment, and the latter considers the risks associated with these events. If a system is found to have safety implications it must be allocated an integrity level reflecting its level of criticality. The safety integrity level will determine the methods of design and implementation to be used for the system. Section 3.5 will therefore discuss what safety integrity level is and how it is assigned. The acceptability of a given level of risk is determined by the benefits associated with the risk, and by the amount of effort that would be required to reduce it. Acceptance of a particular risk is based upon a decision as to whether the risk is *as low as is reasonably practicable* (ALARP). Finally, subchapter 3.7 looks at areas in the standard that have to be fulfilled in order to claim compliance.

3.1 Safety-related system

Before looking at safety-related system, we will discuss the term safety. Definitions of safety vary considerably. The standard IEC 61508, that will be discussed later in this chapter, states this definition:

> *Safety is freedom from unacceptable risk.*

A system is said to be safe if it will not endanger human life or the environment. Computer systems can only influence safety if they are used to control some physical process which can lead to harm. In order to relate the concept of safety to computer systems, the standard introduces the notion that computer systems offer services to the controlled equipment. They operate in order to try to satisfy some goal in the management of the equipment which the computer system is designed to control or influence. IEC uses the term Equipment Under Control (EUC), and defines the term safety-related system as follows:

> *Safety-related system is a designated system that both*
> - *implements the required safety functions necessary to achieve or maintain a safe state for the EUC.*
> - *is intended to achieve, on its own or with other E/E/PE safety-related systems, other technology safety-related systems or external risk reduction facilities, the necessary safety integrity for the required safety functions.*

The term safety-critical system is normally used as a synonym for a safety-related system, but in some contexts it can suggest a system of higher criticality than a safety-related system. In this report safety-critical system and safety-related system are used as synonyms.

Safety-critical systems are a special class of composite system whose development has traditionally relied on close adherence to structured methods, strict quality control and the occasional use of time-consuming and expensive formal methods. Closer inspection, however, reveals that these systems have in fact a limited number of safety-critical aspects or components [42].

Examples of safety-related systems:
- Crane automatic safe load indicator
- Fairground roller-coaster control system
- Fire and gas detection system
- Machinery guard/access interlocking system
- Machinery emergency shutdown
- Process plant emergency shutdown system
- Railway signaling
- Steam boiler controls

Even a safety-critical system can not be absolutely safe. The goal when developing a system is to make it adequately safe for its given role. The standard IEC 61508, that we will explain next, can help in that process.

3.2 Overview of IEC 61508

International standards such as IEC 61508 covers the design and application of safety-related systems. These standards fulfill several important roles [54]:
- Help staff to ensure that a product meets a certain level of quality.
- Help to establish that a product has been developed using methods of known effectiveness.
- Promote a uniformity of approach between different teams.
- Provide guidance on design and development techniques.
- Provide some legal basis in the case of a dispute.

IEC 61508 is a standard for developing safety-critical systems and a framework for developing domain-specific safety standards. It has been approved and published in full. The four first parts of IEC 61508 were published in 1998; the three remaining parts were published in March and April 2000. It is titled “Functional safety of electrical/electronic/programmable safety-related systems”, and is a seven part international standard. The titles of the parts are:
- Part 1: General requirements (IEC 61508-1)
- Part 2: Requirements for electrical/electronic/programmable electronic systems (IEC 61508-2)
- Part 3: Software requirements (IEC 61508-3)
- Part 4: Definitions and abbreviations (IEC 61508-4)

- Part 5: Guidelines on application of Part 1 (IEC 61508-5)
- Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3 (IEC 61508-6)
- Part 7: Bibliography of techniques (IEC 61508-7)

Part 1, 2, 3, and 4, except for the annexes to Part 1, state the definitive requirements of the standard. Part 5, 6, and 7 offers guidance and supplement to the other parts.

The standard is generic, it is not limited to any specific industrial sector or application area. It is intended to be used as a basis for development of more specific standards. Examples of application specific standards derived from IEC 61508 are IEC 61511 for the process sector, IEC 61513 for the nuclear sector and IEC 62061 for the machinery sector. Where application specific standards don't exist, IEC 61508 can be used directly. It can be applied both to systems which operate 'on demand' and to those that are required to operate continuously to maintain a safe state.

IEC 61508 applies to safety-related systems when one or more of such systems incorporate electrical and/or electronic and/or programmable electronic (E/E/PE) devices. It is generically based on and applicable to all E/E/PE safety-related systems irrespective of application. It is recognized that the consequences of failure could also have serious economic implications and in such cases the standard could be used to specify any E/E/PE safety-related system used for the protection of equipment or product [21].

The range of E/E/PE safety-related systems to which IEC 61508 can be applied includes [21]:

- Emergency shut-down systems
- Fire and gas systems
- Turbine control
- Gas burner management
- Dynamic positioning (control of a ship's movement when in proximity to an offshore installation)
- Railway signaling systems (including moving block train signaling)
- Crane automatic safe-load indicators and
- Machinery guard interlocking systems.

Many companies have already adopted this international standard in order to protect persons, equipment and the environment. According to [4], the key objectives of the standard are to:

- Release the potential of the technology to facilitate improvements in both safety and economic performance.
- Enable the technological developments to take place within an overall safety framework.
- Provide a systematic approach to all safety lifecycle activities and all elements of the system including hardware and software.
- Provide a technically sound, systems based approach with sufficient flexibility for the future.

- Provide a risk based approach for the determination of the required performance of safety-related systems.
- Provide a generically based standard which could be used directly by industry but which also facilitate the development of sector standards. A major objective of the publication of IEC 61508 is to enable the development of application sector international standards by the Technical Committees responsible for that particular sector. This should lead to a higher level of consistency (e.g. of underlying principles, technical requirements for a specified performance).
- Provide confidence to users and regulators when using computer-based technology.
- Provide a coherent standard based on common underlying principles which would facilitate:
    - Improved efficiencies in the supply chain for suppliers of subsystems and components to various sectors.
    - Improvements in communication and requirements (i.e. to enable improved clarity of what was required by the system).
    - The development of techniques and measures that could be used across all sectors which could provide high resources gearing.
    - The development of conformity assessment services if stakeholders required this.

IEC 61508 can therefore help to develop a product in a well defined safety-oriented development process that fulfills its safety requirements, reduce cost or time-to-marked, increase confidence within the own organization, and easier get acceptance by customers [5].

The aim is to address all possible causes of dangerous failures. Such failures could arise due to faults in hardware, software in any part of the safety-related system or from human error. Further, faults can be introduced at any stages of the lifecycle of a system, from its initial concept, through design, installation and operation to eventual decommissioning.

The strategy of the standard is first to derive the safety requirements of the safety-related system from a hazard and risk analysis. Thereafter, the safety-related system should be designed to meet those safety requirements, taking into account all possible causes of failure including random hardware faults, systematic faults in both hardware and software, and human factors.

IEC 61508 is based on safety functions. A safety function is an action, which is required to ensure that the risk associated with a particular hazard is tolerable. It is specified in terms of its functionality and its safety integrity [7]. Furthermore, it describes measures and techniques which the safety-related system has to be designed, developed and implemented according to, in order to achieve the necessary safety integrity. Safety functions also describe how the safety integrity of a system has to be estimated. This approach is independent of a specific technology and architecture of the system and its subsystem.

According to the standard it is not valid to assume that, if the EUC and its control systems are built well and are reliable, they will be safe, but it is a good starting point. The safety lifecycle presented in the next section provide a systematic approach to all development activities.

3.3 Safety lifecycle

IEC 61508 uses the safety lifecycle as a framework to structure its requirements. It covers not merely a systems development, but all the principal phases of its existence. The essence is that all activities relating to functional safety are managed in a planned and methodical way. This enables a process of verification whereby a check is made at the conclusion of each phase to confirm that the required outputs have, in fact, been produced and planned. The premise is that such a structured approach will minimize the number of systematic faults that are 'built-in' to the safety-related system [7].

The safety lifecycle also clearly identifies activities related to the planning of the later stages of the development process. These tasks are shown as taking place in parallel to the realization phases of the project. The safety lifecycle covers all aspects of a project, from the conception of a system to its eventual decommissioning. It also considers the impact of modifications during the system's life.

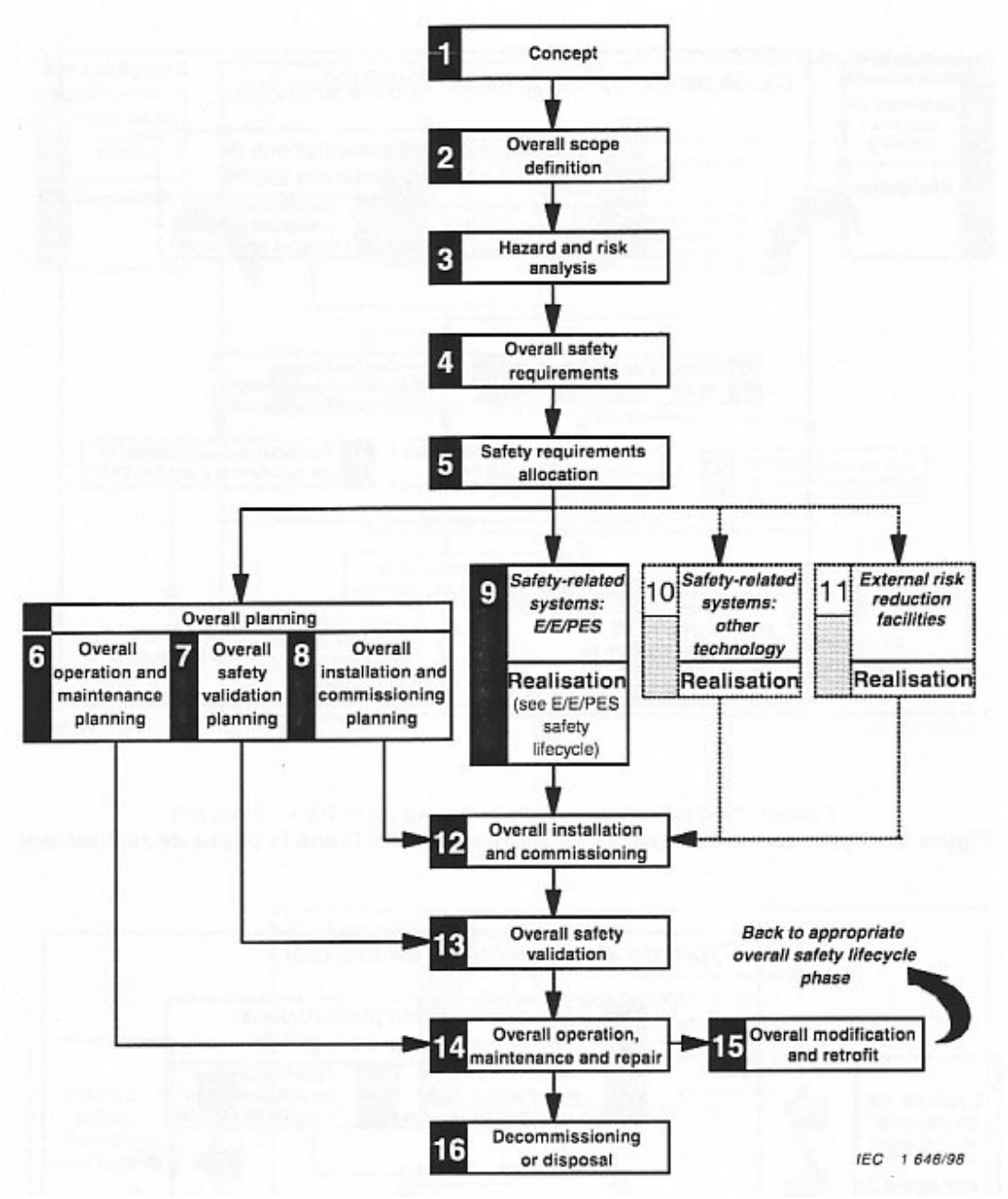


Figure 2 Overall safety lifecycle [25]

The phases of the safety lifecycle are represented by numbered boxes in the diagram (see Figure 2). Each phase has an input, a defined function and an output. The output from one phase represents the input to the next. Verification and assessment take place within each phase to ensure that these activities are performed correctly. For instance, the hazard and risk analysis associated with phase 3 of the model is used within phase 4 to determine the appropriate integrity level for the system. This will in turn determine the form of the other phases of the model, e.g. which methods and techniques that are appropriate. Thus, the model can be used for developing systems of differing levels of integrity, and the

activities represented by the boxes may be very different. The safety of a system is determined not only by its design and development, but also by how it is installed, used and maintained [54]. For this reason an overall strategy for commissioning, operation and maintenance is established at an early stage in the development process, at a time when it can influence the detailed design of the system.

Activities related to the management of functional safety, verification and functional safety assessment are also part of the overall lifecycle, but they are not included in the overall safety lifecycle model in order to reduce its complexity.

3.4 Hazard and risk analysis

Hazard and risk analysis are at the heart of any system safety program. For any system that is safety related, a detailed hazard and risk analysis is required in order to determine an appropriate integrity level for the project. Hazard analysis is concerned not only with the characteristics of the system, but also with details of the design. Its aim is to define the EUC risk, quantitatively or qualitatively, and to determine a 'tolerable' level of risk. The hazard and risk analysis is phase three of the safety-lifecycle - see Figure 2. The result of this phase allows the necessary risk reduction. Hazard and risk analysis involves hazard identification, hazard analysis, and risk assessment. Hazard analysis will normally continue throughout the development process. Redmill [41] stresses the essential for a hazard identification: "The importance of hazard identification cannot be emphasized too strongly – for the risks associated with unidentified hazards will remain unreduced – and care should be taken in cutting corners in carrying it out."

The hazard and risk analysis shall consider the following (IEC 61508-1 7.4.2.10):

- Each determined hazardous event and the components that contribute to it.
- The consequences and likelihood of the event sequences with which each hazardous event is associated.
- The necessary risk reduction for each hazardous event.
- The measures taken to reduce or remove hazards and risks.
- The assumptions made during the analysis of the risks, including the estimated demand rates and equipment failure rates; any credit taken for operational constraints or human intervention shall be detailed.
- References to key information (see IEC 61508-1 clause 5 and annex A) which relates to the safety-related systems at each E/E/PES safety lifecycle phase (for example verification and validation activities).

Hazard analysis is used for [44]:

- Developing requirements and design constraints
- Validating requirements and design for safety
- Preparing operational procedures and instructions
- Test planning
- Management planning

and serves as:

- A framework for ensuing steps

- A checklist to ensure management and technical responsibilities for safety are accomplished.

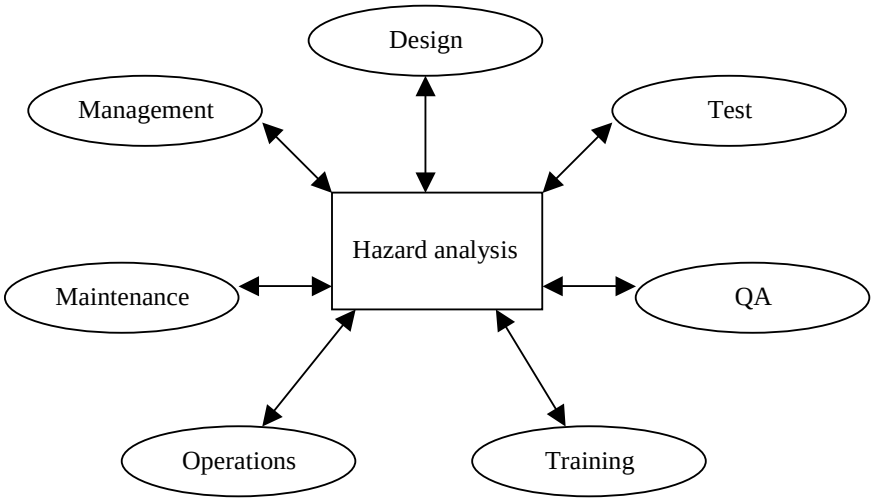


Figure 3 Hazard analysis [44]

Hazard analysis affect, and is affected by all aspects of the development process, see Figure 3. Hazards are managed through identification, evaluation, elimination, and control. System hazards are not failures. Failures may contribute to hazards, but hazards are system states that, combined with certain environmental conditions, cause accidents [45]. Strict definitions of these concepts can be found in Appendix A.

Hazard is not a characteristic of the system or equipment alone. In order to do a hazard and risk analyses, these factors have to be understood; the system, how the system is used, and the system's environment [5].

3.5 Safety Integrity Level

The consequences of failure vary greatly between applications. Therefore the concept of levels of integrity was introduced. Levels of integrity reflect the importance of correct operation. The assigned safety integrity level (SIL) of a project determines the methods of design and implementation used for the system. SILs are defined in terms of failure rates of a function [38]. They depend not only on the failure rates of safety functions, but also on the acceptability of the risks involved [38].

Based on the assessment of the hazard and risk analysis, an appropriate safety integrity level can be allocated. Several factors are affected when SIL is allocated, see Figure 4 for an illustration.

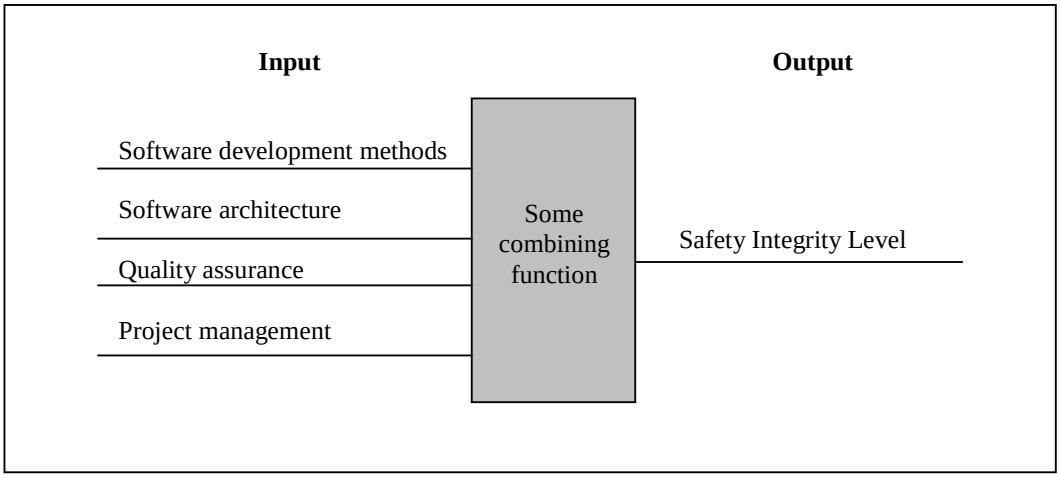


Figure 4 Factors affecting Safety Integrity Levels [6]

A safety integrity level must be allocated to each safety function. The SILs are used for specifying the target level of safety integrity for the safety functions to be implemented by the E/E/PE safety-related systems. The SIL for a safety function is determined by Table 3. The table distinguish between whether the safety integrity requirements is expressed in terms of the average probability of failure on demand, or the probability of a dangerous failure per hour of operation.

Table 3 Safety integrity levels: target failure measures for a safety function [25]

Safety integrity level	Low demand mode of operation (Average probability of failure to perform its design function on demand)	High demand or continuous mode of operation (Probability of a dangerous failure per hour)
4	$\geq 10^{-5}$ to $< 10^{-4}$	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-4}$ to $< 10^{-3}$	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-3}$ to $< 10^{-2}$	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-2}$ to $< 10^{-1}$	$\geq 10^{-6}$ to $< 10^{-5}$

IEC 61508 specifies the requirements to be met for each safety integrity level. Using the process implied by a SIL does not mean that the reliability represented by the SIL has been achieved. Achieving the SIL does not imply that the system is safe, and meeting the requirements of a SIL offers confidence, not proof. SIL provide targets for risk reduction. Although SIL is based on risk, it is not a measure of risk – it is the probability of failure of a system or function.

3.6 ALARP

The ALARP principle is a tool for determining tolerance to risks. IEC 61508 divides the levels of risk into three ranges: unacceptable, acceptable, and ALARP. The uppermost band of this figure represents hazards where the risk is so great that it is considered to be intolerable and can not be justified on any grounds. In contrast, the lowermost band represents hazards where the risk is so small that it can be neglected. In the ARLAP region, between these two, the risk can be acceptable under certain circumstances. The criterion for acceptance of a particular risk is based on a decision as to whether it is "As Low as Reasonably Practicable", (ALARP). This method acknowledges that when reducing process risk there is often a point of diminishing returns. Risks cannot be completely eliminated, even if an infinite amount of money is expended.

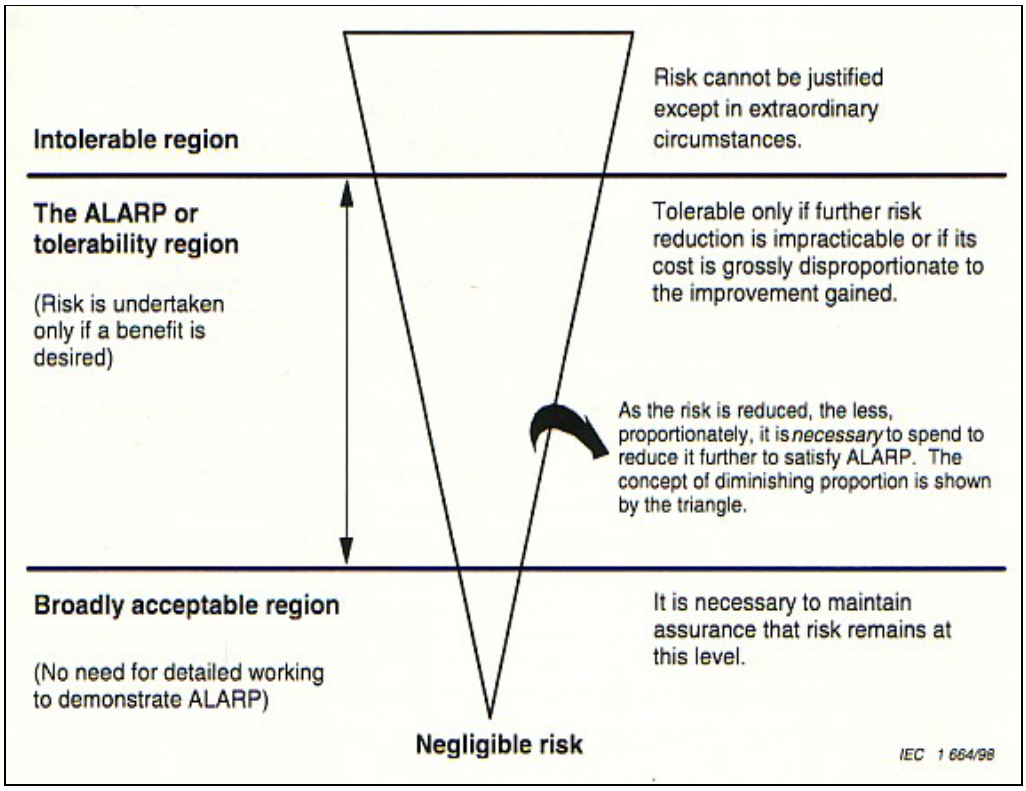


Figure 5 Tolerable risk and ALARP [25]

While the cost versus benefit philosophy makes this an attractive methodology, care must be taken in implementation. The benefit associated with installing or improving a safety-related system design is often a reduction in the number of injuries or fatalities. In order to make sure that the benefits are measured as carefully as the perceived costs, the user of this method must place a quantified value to each potential injury or fatality. Otherwise, an individual project budget would determine the value. This is unacceptable from a corporate risk standpoint [56].

3.7 Compliance

IEC 61508 has three key areas that have to be fulfilled in order to claim compliance with the standard. These areas are; general requirements, techniques and methods, and third party certified components. All the statements below are collected from the IEC website [20].

3.7.1 Requirements

The term *shall* used in a requirement indicates that the requirement is strictly to be followed if conformance to the standard is to be claimed. Where the term *should* (or *it is recommended that*) is used, this indicates that among several possibilities one is recommended as particularly suitable, or that a certain course of action is preferred but not necessarily required. The text in a normative element usually contains both shall and should. In IEC 61508, the following parts contain normative elements: part 1 (excluding annexes); part 2 (including annexes); part 3 (including annexes A and B, excluding annex C); and part 4 (excluding the annex). There are no normative requirements in parts 5, 6 and 7 of the standard. Informative elements of the standard provide additional information intended to assist understanding or use, but with which it is not necessary to conform in order to be able to claim compliance. The text in an informative element cannot contain shall. Notes and footnotes are always informative. In IEC 61508, the following are informative: the annexes of part 1; annex C of part 3; the annex of part 4; and all annexes of parts 5, 6 and 7.

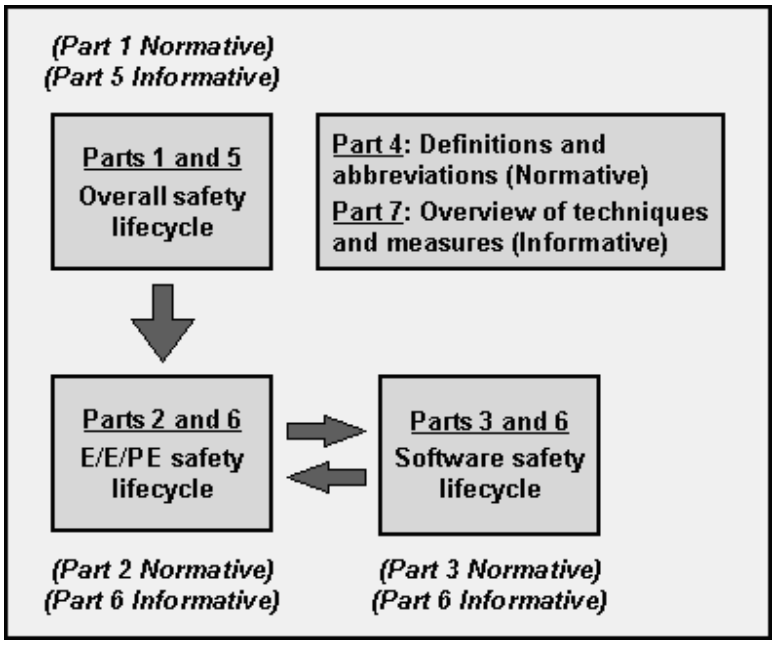


Figure 6 The part structure of IEC 61508 [23]

3.7.2 Techniques and measures

Although all four normative annexes contain recommendations for the use of particular techniques and measures, they differ in what is required for compliance. When a

technique or measure that is highly recommended for the safety integrity level is not used the rationale behind not using it shall be documented in great detail. Annexes A and B of IEC 61508-3 contain the appropriate techniques and measures that shall be selected according to the safety integrity level. Anyone claiming compliance with the standard is required to consider which techniques or measures are most appropriate for the specific problems encountered during the development of each E/E/PE safety-related system. These may include techniques and measures recommended by the standard and may include others; the tables give only recommendations as to which techniques and measures may be appropriate.

Since a large number of factors affect software safety integrity, IEC 61508-3 cannot prescribe generically how to combine techniques and measures in order to guarantee that the required software safety integrity is achieved. The annexes contain a recommendation that the rationale for not following the guidance for highly recommended or not recommended techniques or measures should be detailed during the safety planning, and agreed with the assessor. In both IEC 61508-2 and IEC 61508-3, the choice of techniques for each lifecycle phase needs to be documented (see clause 5 of IEC 61508-1). Other subclauses require some of this documentation to include a justification of the choice of techniques and measures, even if all recommendations are followed. See for example 7.3.2.2 e), 7.4.2.9 of IEC 61508-2, and 7.4.3.2 a) of IEC 61508-3.

3.7.3 Third party certified components

One does not have to use third party certified components in order to comply with IEC 61508. The standard requires that a functional safety assessment is carried out on all parts of the E/E/PE safety-related system and for all stages of the lifecycle (see clause 8 of IEC 61508-1). The level of independence required of the assessor ranges from an independent person in the same organization for safety integrity level 1 to an independent organization for safety integrity level 4. The required level of independence for safety integrity levels 2 and 3 is affected by additional factors including system complexity, novelty of design, and previous experience of the developers. There is also a specific requirement that the assessor shall be considered competent for the activities to be undertaken.

4 Comparison of XP and IEC 61508

Chapter 2 and Chapter 3 discussed Extreme Programming and IEC 61508 separately. In this chapter we perform a comparison of XP and IEC 61508. First we present a high-level comparison, then the individual steps of planning, design, coding, validation and verification, management, and documentation are further analyzed. Each attribute is briefly discussed with respect to XP and IEC 61508, followed by some comments. Attributes which are fundamentally different with respect to XP and IEC 61508, are further discussed in Chapter 5.

Only the most important attributes presented in the high-level comparison will be analyzed later in this chapter.

4.1 High-level

Table 4 shows a top-level comparison between XP and IEC 61508. It provides a framework to zoo in on some low-level issues that will be presented in the following subchapters.

Table 4 High-level comparison of XP and developing according to IEC 61508

	XP	IEC 61508
Development lifecycles	Evolutionary lifecycle, small increments, iterations (1-3 weeks)	V-model (safety lifecycle)
Planning	Short-range	Long-range
Design	Evolutionary design (simple)	Planned design (full up-front)
Integration	Continuous (every few hours)	Infrequently
Releases	Small, frequent (3-6 month)	Big, seldom
Level of scale	Small teams (up to ten developers)	Medium or large teams
Competence of persons	Communication skill	Safety knowledge
Role of developers	Steer process and product, use predefined techniques to do so.	Develop product
Role of managers	Steer process and product, use predefined techniques to do so.	Steer process and product, use predefined techniques to do so (see clause 6).
Role of customers	Provide input to steer process; are involved (on-site).	-
Type of technology	Lightweight technology is preferred.	Certified technology is preferred.
Purpose of development	Product delivery while doing satisfying work.	Develop E/E/PE safety-related system.

Documentation	User stories, tests and code	From all phases of the development cycle
Communication	Person-to-person	Via documentation
Scope	Development (maintenance)	Development, installation, maintenance, and decommission (technical and business processes)

4.1.1 Development cycles

XP: In XP planning, analyzing, designing, testing, and implementation are done in small increments throughout software development.

IEC 61508: The overall safety lifecycle describes required activities associated with safety during the entire lifecycle of the equipment, from the concept phase to the decommissioning phase. The development of the software is structured into defined phases and activities (see Figure 7).

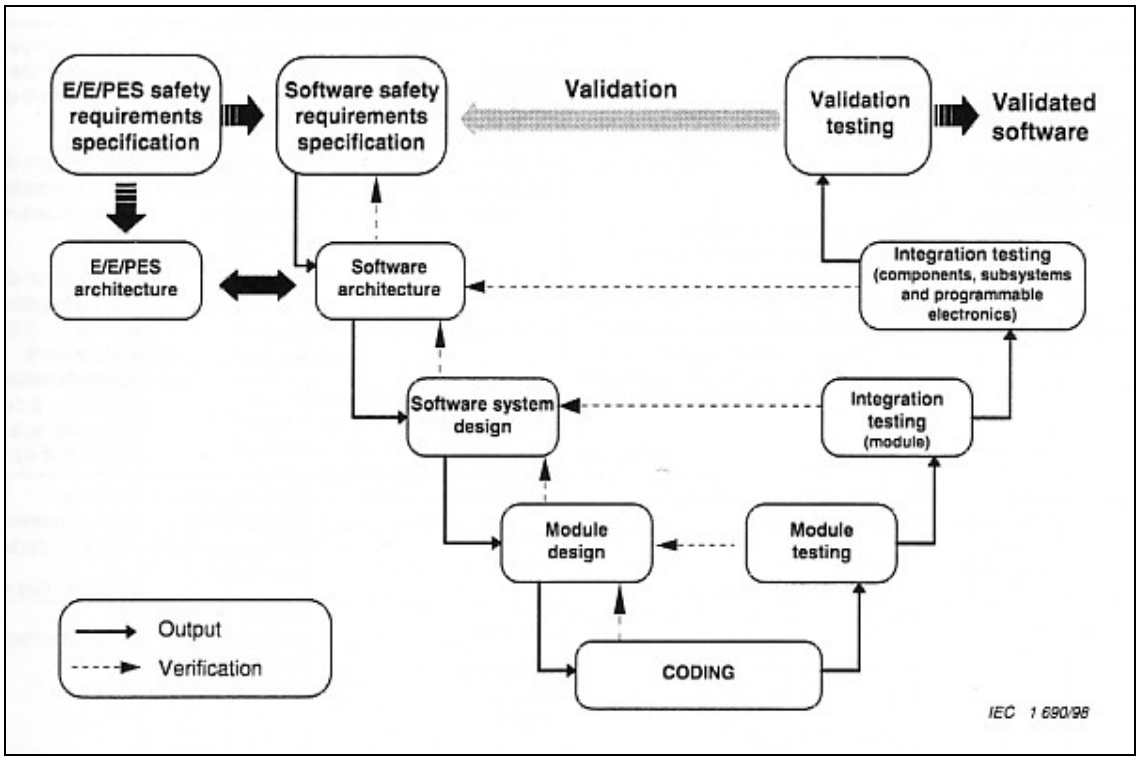


Figure 7 Software safety integrity and the development lifecycle (the V-model) [25]

Comments: The use of an iterative vs. a V-model development model will be discussed further in Chapter 5. Which methods and techniques that are appropriate are discussed in relation to the SIL levels.

4.1.2 Planning

XP: Kent Beck says: "Do only the planning you need for the next release, the end of the next iteration. You can do long-range planning in not great detail [1]." Thus, XP makes only detailed planning for the next release, and spends less time planning the long-range.

IEC 61508: The functional safety planning in IEC 61508 shall define the strategy for the software procurement, development, integration, verification, validation and modification to the extent required by the safety integrity level of the E/E/PE safety-related system (IEC 61508-3 6.2.2).

Comments: The XP practices and IEC 61508 requirements about planning contradict. The safety planning described in the standard is necessary and has to be performed. See Chapter 5 for a further discussion.

4.1.3 Design

XP: Evolutionary design means that the design of the system grows as the system is implemented. The project starts with a simple design that constantly evolves to add needed flexibility and remove unneeded complexity. Design in XP is part of the programming process, and as the program evolves, the design changes. A feeling for the whole is needed up front, but only requirements for the current feature can be clearly defined.

IEC 61508: The planned design approach focus on the big issues before programming. Design techniques are used to work at an abstract level, and design documents are produced. The system should be designed full up-front according to IEC 61508.

Comments: This different is fundamental, and will be discussed further in Chapter 5.

4.1.4 Integration

XP: The system is integrated and build many times a day.

IEC 61508: The system is integrated after the implementation is finished.

Comments: This different is fundamental, and will be discussed further in Chapter 5.

4.1.5 Release

XP: In XP, simple versions of the system are put into production quickly. The customer pick the most valuable stories that make sense together, and let the XP-team implement those and put them into production. New versions are released on a short cycle, new stories are selected and implemented.

IEC 61508: IEC 61508 does not mention releases, but in the view of the safety lifecycle, the releases will be large and seldom, it takes time to make something useful for the customer.

Comments: Like integration, the different practice on release is significant, and will be further discussed in Chapter 5.

4.1.6 Level of scale

XP: XP addresses only small teams, up to ten developers, that must be co-located.

IEC 61508: IEC 61508 does not specify the size of the team, but because of the required competence of persons the addressed team will be medium or large.

Comments: When developing safety-critical systems, every person involved should have the right competence (see below). The safety aspect of the system requires more competence than a traditionally XP team can give. They should for example have safety engineering knowledge appropriate to the technology, and knowledge of the legal and safety regulatory framework. See appendix A in IEC 61508-1 for more details. The scale of the team will therefore be larger than a traditionally XP team, but to make use of XP in the development process the number of developers should not exceed ten persons.

4.1.7 Competence of persons

XP: The value of communication represents the XP belief that communication between project members is important for a successful project, and hence the persons involved should have good communication skills. Instead of specialists (analysts, testers, coders, architects), XP encourages generalists who are handy with all aspects of software development. To pair-program effectively it is recommended that at least one of the programmer is a senior.

IEC 61508: All person involved in safety lifecycle activities, including management activities, should have the appropriate training, technical knowledge, experience and qualifications relevant to the specific duties they have to perform.

Comments: The different focus at competence areas does not exclude each other. A system safety engineer (as a member of the XP team) also requires good communication skills, in addition to knowledge of system safety engineering. A software safety engineer should have good knowledge of both hardware and software. This engineer also needs the ability to communicate with and influence both software engineers and hardware engineers. A software safety engineer is responsible for software safety analyses and for software inputs to system safety analyses. They must participate in design reviews and sit on the configuration control board. The software safety engineer establishes and oversees the audit trails for identified software hazards. The software safety engineer should participate in the integration tests and other activities [48].

4.1.8 Role of developers

XP: Developers have the opportunity to steer the development process, but they have the XP practices as a basis. They can also influence the product through the dialog with the customer.

IEC 61508: The developers responsibility are to develop the product, were they follow the recommendations proposed by the standard or the management.

Comments: To get conformance to IEC 61508, the degrees of freedom of the XP team will have to be reduced. But the standard does not specify all aspect of the development process, and these the developers have the opportunity to steer. E.g. use of pair programming, use of stand-up meetings, and furnishing.

4.1.9 Role of managers

XP: The managers are concerned with the technical execution and evolution of the process, and guide the tracking. The job is to get everybody else making good decisions.

IEC 61508: Manager in a safety-related system shall develop, nurture and maintain a genuine safety culture. Management decision and authority is needed to guide and enforce the use of administrative and technical controls. See IEC 61508-1 clause 6 for a detailed description.

Comments: The role of the managers is not in contradiction, most of their job is to motivate others, and take important decisions on behalf of the team.

4.1.10 Role of customers

XP: Representatives from the customer have the responsibility for the business decisions that take place all through the life of a project. Therefore, the customer is a part of the XP team, and for best result they sit with the rest of the team and are available full-time to answer questions.

IEC 61508: The role of customers is not mentioned in the standard. It is unlikely that the customer is involved in the development.

Comments: Having an on-site customer in the team is not in contradiction with the requirements in IEC 61508.

4.1.11 Type of technology

XP and IEC 61508 have a different understanding of the term *technology*. XP focus on the process while IEC 61508 concentrates on the realization of the system.

XP: The choice of technology is a business decision, but one that must be taken with input from the developer. XP prefer lightweight technology. In XP, they use technology only insofar as it fits the framework of predefined techniques. The technology used

should not have an inherently exponential cost curve, the code should be clean and simple [1].

IEC 61508: IEC 61508 applies to safety-related systems when one or more of such systems incorporate electrical and/or electronic and/or programmable electronic (E/E/PE) devices.

Comments: The type of methods used in XP can be in contradiction with the technology used in safety-critical systems. Especially avoiding technology that have an exponential cost curve. “If the cost of change rose slowly over time, you would act completely different from how you do under the assumption that costs rise exponentially. You would make big decisions as late in the process as possible, to defer the cost of making the decisions and to have the greatest possible chance that they would be right” [1].

Employing complex technology in safety-related system has several disadvantages. For example, the implementation of complex technology requires a higher level of competence at all stages, from specification up to maintenance and operation. The use of other, simpler, technology solutions may be equally effective and may have several advantages of the reduced complexity [25]. These thoughts correspond with XP view of simplicity.

In the context of low complexity E/E/PE safety-related systems, certain requirements specified in IEC 61508 can be unnecessary, and exemption from compliance with such requirements is possible. However, dependable field experience must exist which provides the necessary confidence that the required safety integrity can be achieved.

4.1.12 Purpose of development

XP: Making software development a humane experience is a key motivation of XP. Doing satisfying work is equally important as delivering a product.

IEC 61508: IEC 61508 provides a method for the development of the safety requirements specification necessary to achieve the required functional safety for E/E/PE safety-related systems.

Comments: Doing satisfying work is also possible when developing safety-critical systems, and therefore XP and IEC 61508s purpose of development can be united.

4.1.13 Documentation

XP: XP uses face-to-face human communication in place of written documentation wherever possible. User stories, tests and code will be produced when following the XP practices. If the team need needs additional documents they should produce them.

IEC 61508: IEC 61508 has several requirements to documentation (IEC 61508-1 5.2):

- The documentation shall contain sufficient information, for each phase of the overall, E/E/PES and software safety lifecycles completed, necessary for effective performance of subsequent phases and verification activities.
- The documentation shall contain sufficient information required for the management of functional safety.
- The documentation shall contain sufficient information required for the implementation of a functional safety assessment, together with the information and results derived from any functional safety assessment.

Comments: XP and IEC 61508 have quite different view on the amount of documentation that is necessary to produce. But XP gives the opportunity to produce documents that are needed, and therefore they are not completely in contradiction. See section 4.1.13 and Chapter 5 section 5.15 for a further discussion.

4.1.14 Communication

XP: Communication is one of XPs four values that are needed for a successful project. Many of the XP-practices can not be done without verbal communication.

IEC 61508: The standard impose the engineers to make documents to communicate.

Comments: XP often uses verbal-communication instead of documentation. Safety related activities also require verbal-communication, but in addition documents are written. See section 4.1.13 for a further discussion.

4.1.15 Scope

XP: XP covers the development process for building software with vague and rapidly changing requirements. Kent Beck [1] notes that “maintenance is really the normal state of an XP project”.

IEC 61508: IEC 61508 can be applied to the implementation, operation including maintenance, installation, and decommission of any safety-related control or protection system based on electrical/electronic/programmable electronic technology.

Comments: IEC 61508 has a bigger scope than XP. Chapter 5 gives a proposal on how XP can replace the development process and maintenance process described in IEC 61508.

4.2 Planning

Next follows a comparison of XP and IEC 61508 with respect to selected planning aspect. Some comments for each attribute follow.

Table 5 Comparison of XP and IEC 61508 with respect to planning

	XP	IEC 61508
Project plan	Rough plan that will be refined.	Out of scope
Planning phase	Exploration phase, iteration planning	Overall planning
Focus on environment	None	High
Lifecycle requirements	No	Yes
Tool selection	Integration, build, test	Qualified, suitable for purpose

4.2.1 Project plan

XP: Planning in XP is an activity in which the development team, manager and customer decide on what to do in each release and iteration. The planning game is used to create a rough plan quickly and refine it later as things become clear.

IEC 61508: A project plan is not in the scope of the standard, and is therefore not specified.

Comments: XPs plan is not in contradiction with IEC 61508s requirements because the standard does not specify a project plan.

4.2.2 Planning phase

XP: A release starts with an exploration phase, in which customer and developer discuss what the system should do. Iteration planning starts by asking the customer to pick the most valuable stories. The team breaks the stories down into tasks. Next the programmers signs up for the tasks they want to be responsible for implementing. The programmer estimates the task in ideal programming days.

IEC 61508: The overall planning in IEC 61508 includes operation and maintenance planning, safety validation planning, and installation and commissioning planning.

Comments: The exploration phase in XP might be considered a parallel to requirements analysis in traditionally software engineering. In this phase, the developers have the freedom to experiment with the solutions they will be proposing and implementing. The overall planning in IEC 61508 covers future planning aspects as opposed to XP short planning aspect. Planning in XP and in IEC 61508 have different focus, and does not contradict.

4.2.3 Focus on environment

XP: XP has no specific focus on the environment except that it should be possible to realistically test software.

IEC 61508: The concept- and overall scope definition-phase focus on the environment. The objective of the concept-phase is to develop a level of understanding of the equipment under control and its environment sufficient to enable the other safety lifecycle activities to be satisfactorily carried out. One of the objectives of the overall scope definition phase is to specify the scope of the hazard and risk analysis, for example environmental hazards.

Comments: The focus on environment does not contradict. IEC 61508 requires also that it should be possible to realistically test software.

4.2.4 Lifecycle requirements

XP: XP recommends the developer to follow the XP practices, but let them change them if something does not work.

IEC 61508: The objective of the software safety lifecycle requirements in IEC 61508 is to structure the development of the software into defined phases and activities. This structure and its related requirements are IEC 62508-3 7.1.2.1-7.1.2.8.

Important issues are:

- A safety lifecycle shall be selected and specified
- Quality and safety assurance procedures shall be included in the safety lifecycle
- Each phase of the safety lifecycle shall be divided into elementary activities with the scope, input and output for each phase
- It is acceptable to tailor the depth, number and work-size of the phases of the V-model
- It is acceptable to order the software project differently to the organization of this standard provided that the objectives and requirements of *clause 7 Software safety lifecycle requirements* are met.
- For each lifecycle phase, appropriate techniques and measures shall be used
- The result shall be documented
- If, at any stage of the software safety lifecycle, a change is required pertaining to an earlier lifecycle phase, then that earlier safety lifecycle phase and the following phases shall be repeated

Comments: In principle the lifecycle requirements do contradict, but both XP and IEC 61508 are to a certain degree receptive to changes. IEC 61508, does, however, require that the requirements in the standard must be fulfilled, but these requirements do not completely restrict the development process. In Chapter 5 a model different from the software safety lifecycle in IEC 61508 are proposed.

4.2.5 Tool selection

XP: It is important to XP to have tools that support a fast integration/build/test cycle [1]. See section 4.2.5 for a description of testing framework.

IEC 61508: IEC 61508 provides requirements for support tools such as development and design tools, language translators, testing and debugging tools, and configuration management tools. IEC 61508 gives detailed guidance on the tools that are appropriate for the various phases of a project, for systems of various levels of integrity. The selection of development tools will depend on the nature of the software development activities and the software architecture. The tool selection must be performed according to IEC 61508-3 7.7.2.7:

- All equipment used for validation shall be qualified according to a specification traceable to an international standard (if available), or to a national standard (if available), or to a well-recognized procedure.
- Equipment used for software validation shall be qualified appropriately and any tools used, hardware or software, shall be shown to be suitable for purpose.

A suitable set of integrated tools, including languages, compilers, configuration management tool, and when applicable, automatic tools, shall be selected for the required safety integrity level. The availability of suitable tools (not necessarily those used during initial system development) to supply the relevant services over the whole lifetime of the E/E/PE safety-related system should be considered. (IEC 61508-3 7.4.4.2)

Comments: XP accept almost every tool, and does not contradict with IEC 61508s requirements with respect to tools. We recommend using the testing framework developed for unit testing.

4.3 Design

Beck [1] describes a good design as follows:

- It organizes the logic so that a change in one part of the system does not always require a change in another part of the system.
- It ensures that every piece of logic in the system has one and only one home.
- It puts the logic near the data it operates on.
- It allows the extension of the system with changes in only one place.

An ideal specification according to IEC 61508 should have a number of characteristics, including that it should be [54]:

- Correct
- Complete
- Consistent
- Unambiguous

Table 6 gives a comparison of XP and IEC 61508 with respect to design. Some comments for each attribute follow.

Table 6 Comparison of XP and IEC 61508 with respect to design

	XP	IEC 61508
Approaches to discuss requirements	Prototype	Formal textual requirements
Architecture	System metaphor	Specific configuration of hardware and software elements in a system (IEC 61508-4 3.3.5).

4.3.1 Approaches to discuss requirements

XP: XP prefers using prototypes to discuss requirements and ensure timely feedback from customers.

IEC 61508: IEC 61508 uses formal textual requirements. The specification of the requirements for software safety shall, according to IEC 61508-3 7.2.2.3, be sufficiently detailed to allow the design and implementation to achieve the required safety integrity, and to allow an assessment of functional safety to be carried out.

Comments: The different approach to discuss requirements will be discussed further in Chapter 5.

4.3.2 Architecture

XP: In XP, part of the architecture is captured by the system metaphor. A good metaphor can tell the team how the system works. The metaphor guides all development with a simple, shared story of how the overall system works. The first iteration puts the architecture in place.

IEC 61508: The software supplier and/or developer shall establish the proposed software architecture design, and a description of the software architecture design shall be detailed. IEC 61508-3 subclause 7.4.3.2 state the requirements of the description. The architecture shall be unambiguously defined.

Comments: The system metaphor in XP will be too vague when developing safety-critical system. A further discussion can be found in Chapter 5.

4.4 Coding

Table 7 gives a comparison of XP and IEC 61508 with respect to coding and coding methods. Some comments for each attribute follow.

Table 7 Comparison of XP and IEC 61508 with respect to coding

	XP	IEC 61508
Refactoring	Yes	No
Programming language	Smalltalk, Java	See comments in section

		4.4.2.
Coding standard	Yes	Yes
Reuse of software	-	Attractive
Redundant source code	Eliminate	-
Method for writing production code	Pair programming	-

4.4.1 Refactoring

XP: XP recommends the programmer to refractor to remove duplication, improve communication, simplify, or add flexibility. The program is restructured, but its behavior is not changed. XP use refactoring as an alternative to up-front design. See Chapter 2 for a more detailed description of refactoring.

IEC 61508: It should not be necessary to refractor when developing according to the software safety lifecycle.

Comments: The developer of a safety-critical system will have to check whether the refactoring does change the code in such a way that the system can become unsafe. Extensive use of refactoring in XP can easily result in an unsafe system. Chapter 5 discusses this further.

4.4.2 Programming language

XP: "XP assumes that the development team makes use of modern development environment (Smalltalk, Java), and aims at taking maximal advantage of the resulting benefits" [11].

IEC 61508: A consideration when selecting a programming language for safety-related system is the portability of the code, that there are no differences between the development on environment and on the target system.

In safety-critical systems, the choice of programming language is of great significance. The functional characteristics, the availability and quality of the support tools, and the expertise available within the development team must be considered before choosing a language.

According to IEC 61508-3 subclause 7.4.4.3 the selected programming language shall:

- Have a translator/compiler which has either a certificate of validation to a recognized national or international standard, or it shall be assessed to establish its fitness for purpose.
- Be completely and unambiguously defined or restricted to unambiguously defined features.
- Match the characteristics of the application.
- Contain features that facilitate the detection of programming mistakes.
- Support features that match the design method.

Comments: XP wants to use modern, high level development language. One advantage of using high level languages is that they make the end code readable, and hence easier to analyze. Therefore will first investigate whether it is possible to develop safety-related system using such languages. Kent Beck's Testing Framework is widely available for Smalltalk, Java and C++ [27], and therefore we will look at these languages first.

The main problem with Java in safety-critical systems will probably be the extensive use of dynamic memory management and the garbage collection, which will make both maximum memory usage and worst-case response times hard to predict [19]. In addition, Java is based on libraries that are not certificated.

Smalltalk will not be a suitable programming language for implementing a safety-critical system. Smalltalk compilers are free to cheat, and Smalltalk system play games with integers [52]. That is not acceptable when implementing safety-critical code.

All SIL highly recommend a strongly typed programming language. C++ have been used with success both in XP [55] and in safety-critical systems. Although an object-oriented approach offers many potential benefits, some of the features associated with OOD languages must be avoided when developing safety-critical systems. One such problem area is linked with the use of dynamic dispatching, which requires that certain operations are determined at runtime. This feature causes great problem for verification and validation of the system's operation. To reduce the probability of introducing programming faults and increase the probability of detecting any remaining faults we recommend using a subset of C++ when developing safety-related system. The language is examined to determine programming constructs which are either error prone or difficult to analyze. A language subset is then defined which excludes these constructs. Before any decisions are made one should discuss a suitable programming language for each individual system.

To be able to make the most of XPs methods, it makes a different which programming language is used. SIL 4, and in some cases SIL 3, may require other programming languages that does not fit to XP. In those cases we recommend to follow IEC 61508.

4.4.3 Coding standard

XP: XP requirements to coding standard are: "call for the least amount of work possible, consistent with the Once and Only Once rule (no duplicate code). The standard should emphasize communication. Finally, the standard must be adopted voluntarily by the whole team" [1].

IEC 61508: The coding standard shall according to IEC 61508-3 requirement 7.4.4.5 and 7.4.4.6:

- Be reviewed as fit for purpose by the assessor.
- Be used for the development of safety-related software.
- Specify good programming practice.
- Proscribe unsafe language features (for example, undefined language features, unstructured designs, etc.). Specify procedures for source code documentation. As

a minimum, the following information should be contained in the source code documentation:
- Legal entity (for example company, author(s), etc.)
- Description
- Inputs and outputs
- Configuration management history

Comments: IEC 61508s requirements about coding standard are not in contradiction with XP since XP does not require a specific coding standard.

4.4.4 Reuse of software

XP: XP does not mention reuse of software. XP says that the programmer should design and code *the simplest thing that could possibly work*, it is possible that this does include reuse of software.

IEC 61508: "Because of the high cost of software development, particularly for critical applications, the reuse of existing software from other projects is commercially very attractive" [54]. IEC 61508 is open for the possibility to use previously developed software.

Comments: Reuse of software is a possibility for both XP and IEC 61508.

4.4.5 Redundant source code

XP: XP insists on stating everything "once and only once" and therefore tries to fully eliminate redundant code. The redundancy is removed by refactoring. When the refactoring is done throughout the entire lifecycle, time is saved and quality increased [59]. See Chapter 2 section X for a description of refactoring.

IEC 61508: All methods of fault tolerance are based on some form of redundancy. This involves having a system which is more complex than that needed simply to perform the required task [54]. IEC 61508 does not mention use of redundant source code.

Comments: Redundancy can increase reliability and reduce failures. However, it assumes a model of random wear out. It is not so effective at common-cause or common-mode failures, which may affect all redundant parts equally. Redundancy can also add so much complexity to the system (to coordinate the redundant components) that the complexity causes failures. Certainly, redundancy has its place, and it can be useful in reducing hardware failures. Therefore we recommend using redundancy in hardware, and not in the software part of the system.

4.4.6 Method for writing production code

XP: The responsible programmer finds a partner to write the production code with.

IEC 61508: IEC 61508 does not state any method for writing production code.

Comments: It is possible to pair-program safety-critical code because the standard does not state any method. Advantages when using pair programming will be discussed in Chapter 5.

4.5 Validation and verification

Validation, and verification are important activities when developing software. Because of the similarity between these activities, we have placed them in the same subchapter. Table 8 gives a comparison of XP and IEC 61508 with respect to validation, and verification. Some comments for each attribute follow.

Table 8 Comparison of XP and IEC 61508 with respect to validation, and verification

	XP	IEC 61508
Quality assurance	Yes	Yes
Safety assurance	No	Yes
Static testing	Yes, source code review	Yes, e.g. reviews, inspections and design walkthroughs.
Unit/module testing	Yes	Yes
Integration testing	Yes	Yes
Acceptance testing	Yes	Yes
Other tests	Any test	Performance testing, probabilistic testing
Validation	Not in scope of XP	Yes
Verification	Yes	Yes
Certification	Some process	Not in the scope of the standard

4.5.1 Quality assurance

XP: Acceptance tests are run often in a XP project and the result is published. The work products and services are therefore objectively evaluated. Quality assurance (QA) is an essential part of the XP process. In some project QA is done by a separate group, while in others QA will be integrated into the development team. External forces (customers, managers) get to pick the values of any three of the variables – cost, time, quality, and scope. The development team gets to pick the value of the fourth variable.

IEC 61508: Quality assurance procedures are integrated into safety lifecycle activities. Examples of such activities are verification activities and assessment.

Comments: Both XP and IEC 61508 focus on quality assurance, but the standard are more concerned about having documentation of the process. In addition IEC 61508 carries out more quality assurance procedures than XP.

4.5.2 Safety assurance

XP: XP has not specific safety assurance.

IEC 61508: Safety assurance procedures are integrated into the safety lifecycle activities. Examples of such activities are hazard and risk analysis, and software safety validation.

Comments: IEC 61508s requirement on safety assurance can not be said to be in contradiction with XP because XP does not require that assurance. See Chapter 5 for a discussion of safety focus and software development.

4.5.3 Static testing

Static testing investigates the characteristics of a system or component without operating it. When code is subject to static testing, the structure and properties of the software are studied. This testing, which is also called static code analysis, is "static" in the sense that the code is not executed but is simply analyzed [54].

XP: XP performs static testing by code reviews. These reviews do not have to be formal – pair programming provides continuous code reviews through the entire development process.

IEC 61508: The source code shall be verified by static methods to ensure conformance to the specified design of the software module, the required coding standard, and the requirements of safety planning. The static methods can be inspection, review, formal proof, and walkthroughs.

Comments: IEC 61508 suggests more testing than XP, see Table 9 for recommendation of techniques.

Table 9 Static testing technique recommendations for SILs

Technique	Ref	SIL 1	SIL 2	SIL 3	SIL 4
Formal proof	Table A.9 IEC 61508 -3	---	R	R	HR
Static analysis	Table A.9 IEC 61508 -3	R	HR	HR	HR
Boundary value analysis (static analysis)	Table B.8 IEC 61508 -3	R	R	HR	HR
Checklists (static analysis)	Table B.8 IEC 61508 -3	R	R	R	R
Control flow analysis (static analysis)	Table B.8 IEC 61508 – 3	R	HR	HR	HR
Data flow analysis (static analysis)	Table B.8 IEC 61508 – 3	R	HR	HR	HR
Error guessing (static analysis)	Table B.8 IEC 61508 –	R	R	R	R

	3				
Fagan inspections (static analysis)	Table B.8 IEC 61508 – 3	---	R	R	HR
Sneak circuit analysis (static analysis)	Table B.8 IEC 61508 – 3	---	---	R	R
Symbolic execution (static analysis)	Table B.8 IEC 61508 – 3	R	R	HR	HR
Walkthroughs/design reviews (static analysis)	Table B.8 IEC 61508 -3	HR	HR	HR	HR

Explanations of the recommendations [25]:
--- : The technique has no recommendation for or against being used.
R : The technique is recommended for this safety integrity level as a lower recommendation to a HR recommendation.
HR: The technique is highly recommended for this safety integrity level. If this technique is not used then the rationale behind not using it should be detailed during the safety planning and agreed with the assessor.

4.5.4 Unit/module testing

XP: Unit testing in XP means testing a unit of code. A unit of code is generally a class in an object-oriented system. It could, however, also be a component or any other piece of related code [55].

Unit test are small, take a white box view on the code, and include a check on the correctness of the result obtained, comparing actual result with the expected ones. Tests are an explicit, but not integrated part of the code, and are put under revision control. Tests does not interact with each other, they are isolated and automatic.

An XP programmer writes a test under the following circumstances [1]:

- If the interface for a method is at all unclear.
- If the interface is clear, but the programmer imagines that the implementation will be the least bit complicated.
- If the programmer thinks of an unusual circumstance in which the code should work as written, he writes a test to communicate the circumstance.
- If the programmer finds a problem later, he writes a test that isolates the problem.
- If the programmer is about to refractor some code, and he are not sure how it is suppose to behave, and there is not already a test for the aspect of the behavior in question, he writes a test first.

IEC 61508: A software module is according to IEC 61508-4 a construct that consists of procedure and/or data declarations and that can also interact with other such constructs.

The requirements for module testing according to IEC 61508 are:

- Each software module shall be tested as specified during software design.
- The test shall show that each software module performs its intended function and does not perform unintended functions.
- The results of the software module testing shall be documented.
- The procedures for corrective action on failure of test shall be specified.

Comments: Both XP and IEC 61508 test each software module, and the test shows that the module performs its intended function. The unit testing in XP can also easily include tests that show that each software unit does not perform unintended functions. IEC 61508 also have requirements for documentation of the results, and procedures for corrective actions.

4.5.5 Integration testing

XP: Testing in XP is typically done using a testing framework such as *Junit* developed by Gamma and Beck [3]. Junit supports Java, but similar testing frameworks are developed for Smalltalk and C++. The framework automatically invokes all test methods of a test class, and collects test cases into test suites. Test results can be checked by invoking any of the assert methods of the framework with which expected values can be compared to actual values. Testing success is visualized through a graphical user interface showing green bar as the tests progress, as soon as a test fail, the bar becomes red [11].

IEC 61508: The integration tests shall ensure that the software satisfies the specification of requirements for software safety at the required safety integrity level. Software integration tests shall be specified concurrently during the design and development phase (IEC 61508-3 7.4.8.1). According to IEC 61508-3 subclause 7.4.8.2 integration testing shall specify:

- The division of the software into manageable integration sets.
- Test cases and test data.
- Types of tests to be performed.
- Test environment, tools, configuration and programs.
- Test criteria on which the completion of the test will be judged.
- Procedures for corrective action on failure of test.

Comments: The integration procedure for XP and IEC 61508 are almost equal. The standard requires, in addition to XPs requirements, procedures for corrective action.

4.5.6 Acceptance test

XP: At the beginning of an iteration in XP, the customers think about what would convince them that the stories for an iteration are completed. These thoughts are converted into system tests. These tests accumulate the customers' confidence in the correct operation of the system.

Acceptance tests are created from user stories, as black box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests have the highest priority. Acceptance tests are also used as regression tests prior to a production release.

IEC 61508: Acceptance test is within the scope of IEC 61508, but the standard does not specify how it should be organized.

Comments: Both XP and IEC carry out acceptance testing. IEC 61508 has no restrictions to the executions or constructions of the tests, and therefore acceptance testing does not contradict.

4.5.7 Other tests

XP: If other tests are needed in an XP project, any suitable test can be written in addition to unit and acceptance tests [1].

IEC 61508: Annex A in IEC 61508-3 mentions performance testing, probabilistic testing as examples of other tests.

Comments: Other tests do not contradict, both XP and IEC 61508 are receptive to using other test.

4.5.8 Validation

XP: Validation is not with in the scope of XP, it is up to the customer to find out whether the system is appropriate for its purpose.

IEC 61508: The software safety validation must provide results according to IEC 61508-3 7.7.2.8. Testing shall be the main validation method for software, animation and modeling can be used to supplement the validation activities.

Comments: IEC 61508s requirements about validation are fundamentally different from what is required in XP, since XP does not require any validation at all. This difference will be further discussed in Chapter 5.

4.5.9 Verification

XP: The source code will be continually reviewed through pair programming. XP runs at least three test; unit tests, integration tests, and acceptance tests. The acceptance tests are seen as the most important once, since it is the customer's verification of the system. As soon as the acceptance tests are ready and the tasks for a story are complete, the acceptance tests are run to verify that the story works [1].

IEC 61508: The verification requirements that shall be met for each overall safety lifecycle phase are specified in 7.18 (IEC 61508-1 7.1.4.8).

The following verification activities shall be performed (IEC 61508-3 7.9.2.7):

- Verification of software safety requirements (see 7.9.2.8).
- Verification of software architecture (see 7.9.2.9).
- Verification of software system design (see 7.9.2.10).
- Verification of software module design (see 7.9.2.11).
- Verification of code (see 7.9.2.12).
- Data verification (see 7.9.2.13).
- Software module testing (see 7.4.7).
- Software integration testing (see 7.4.8).
- Programmable electronic integration testing (see 7.5).
- Software safety requirements testing (software validation) (see 7.7).

IEC 61508 verification is done by review, analysis and/or tests. Selection of techniques and measures for verification, and the degree of independence for the verification activities, will depend upon a number of factors, including:

- Size of the project.
- Degree of complexity.
- Degree of novelty of the design.
- Degree of novelty of the technology.

Comments: The verification activities in IEC 61508 are comprehensive. XPs verification activities are a subset of the standard's verification activities. The verification of requirements, architecture, and design are not present in XP. See Chapter 5 for a more comprehensive discussion of the verification activities.

4.5.10 Certification

XP: Before the software is ready to go into production there will be some process for certification. New tests can be necessary to prove that the software does what it is suppose to do. Parallel testing is often applied at this stage [1].

IEC 61508: Certification is not in the scope of the standard.

Comments: Certification does not contradict the standard since it is not in the scope of IEC 61508.

4.6 Management

The management is of great importance regarding the result of the project. Table 10 gives a comparison of XP and IEC 61508 with respect to management. Some comments for each attribute follow.

Table 10 Comparison of XP and IEC 61508 with respect to management

	XP	IEC 61508
Responsibility	Allocating resources.	Specify all management and technical activities to achieve and maintain the required functional safety.
Roles for people	Customer, tester, coach, consultant, big boss	Developer of the E/E/PES, developer of the software, supplier, user, hazop and risk analysis people, assessment people, domain experts, management people

4.6.1 Responsibility

XP: The managers job is to run the Planning Game, to collect metrics pertaining to project management and control, to make sure that the metrics are seen by those whose work is being measured, and occasionally to intervene in situations that can not be resolved in a distributed way [1].

IEC 61508: Specify all management and technical activities to achieve and maintain the required functional safety (IEC 61508-1 6.2.1)

Comments: The basic XP management tool is the metrics. The role of management is crucial to the effective use of the standard. IEC 61508 does not prescribe exactly what should be done in any particular case. Top management's participation in safety issues is the most effective activity in controlling risk and reducing accidents. Upper management also needs to assign capable people to the system safety effort and give them appropriate objectives and resources. Appropriate organization structures must be set up to ensure that the person responsible for the safety effort has the authority, as well as the responsibility, to ensure system safety. Lastly, upper management must be responsive to initiatives by others [48].

The management responsibilities include setting policy and defining goals. Management must define responsibility, fix accountability, and grant authority. These three properties must be distributed so that people responsible and accountable for system safety have the authority and resources to affect the design and contribution of the system. Communication channels must exists to collect information necessary for safety engineers to perform their tasks, and the system safety effort must be able to disseminate information in such a way as to affect the system designers [48].

4.6.2 Roles for people

XP: Roles in XP: programmer, customer, tester, tracker, coach, consultant, and big boss [1].

IEC 61508: IEC 61508 mention the following roles: developer of the E/E/PES, developer of the software, supplier, user, hazop and risk analysis people, assessment people, domain experts, management people.

Comments: All the roles specified in XP can be found in IEC 61508, and there is therefore no role conflict between XP and IEC 61508.

4.7 Documentation

As we could see from the comments in section 4.1.13, XP and IEC 61508 have different view on what is necessary to document and to what extent. This subchapter discusses further the views XP and IEC 61508 have on documentation.

4.7.1 XP

XP has often been believed to drop documentation and traceability altogether, which is not true. The unit tests serve as excellent documentation. A complete set of tests for a class is often a good starting point for someone looking at that class for the first time and attempting to discover what it does [55].

"Documents can always be produced in the same manner as code is produced, which means that any demands for specific documents are captured as user stories" [55].

4.7.2 IEC 61508

IEC 61508 specifies documents to be produced during the development. Table 11 shows the objectives for each document.

Table 11 Objectives for documents produced according to the overall safety lifecycle

Document	Reference to IEC 61508-1	Objectives
Concept	7.2.2.6	Understanding of the EUC and its environment.
Scope definition	7.3.2.5	Determine the boundary of the EUC and the EUC control system, in addition to specify the scope of the hazard and risk analysis.
Hazards and risk analysis	7.4.2.11	Determine the hazards and hazardous events of the EUC and the EUC control system, determine the event sequences leading to the hazardous events, and determine the EUC risks associated with the hazardous events.
Safety requirements specification	7.5.2.7	Develop the specification for the overall safety requirements, in terms of the safety functions requirements and safety integrity requirements, for the E/E/PE safety-related system, other technology safety-related

		systems and external risk reduction facilities, in order to achieve the required functional safety. Allocate the safety functions, contained in the specification for the overall safety requirements, to the designated E/E/PE safety-related systems, other technology safety-related systems and external risk reduction facilities. Allocate a safety integrity level to each safety function.
Overall operation and maintenance plan	7.7.2.1	Ensure that the required functional safety is maintained during operation and maintenance.
Safety validation plan	7.8.2.2	Facilitate the overall safety validation of the E/E/PE safety-related systems.
Installation plan and commissioning plan	7.9.2.3	Develop plans for the installation and commissioning of the E/E/PE safety-related systems in a controlled manner, to ensure that the required functional safety is achieved.
Modification and retrofit	7.16.2.4	Develop a plan for the modification and retrofit of the E/E/PE safety-related systems to ensure that functional safety for the E/E/PE safety related system is appropriate, both during and after the modification and retrofit phase has taken place.
Decommissioning and disposal impact plan	7.17.2.2	Ensure that the functional safety for the E/E/PE safety-related systems is appropriate for the circumstances during and after the activities of decommissioning or disposing of the EUC.
Verification plan and verification report	7.18.2.1, 7.18.2.4	Demonstrate, for each phase of the overall, E/E/PES and software safety lifecycles (by reviews, analysis and/or tests), that the outputs meet in all respects the objectives and requirements specified for the phase.
Functional safety assessment plan	8.2.8	Investigate and arrive at a judgment on the functional safety achieved by the E/E/PE safety-related systems.

4.7.3 XP versus IEC 61508

Table 12 shows the documentation structure for XP and software safety lifecycle presented in IEC 61508. Some comments for each phase follow.

Table 12 Documentation structure for XP and software safety lifecycle

Phase	XP	IEC 61508
Requirements	User stories	Specification
Architecture	System metaphor	Description
Design	Source code	System and module specification
Coding	Source code and coding standards	Source code and coding standards
Review	-	Code review report
Testing	Unit test and acceptance test	Module test, integration tests (module, system, architecture)
Validation	-	Plan and report
Verification	-	Plan and report
Assessment	-	Plan and report
Modification	-	Instruction, request, report and log

4.7.3.1 Requirements

XP: The first decisions to make about an XP project are what it could do and what it should do first. These decisions are typically the province of analysis. The overall analysis is described as stories.

Each user story must be business-oriented, testable, and estimatable. User stories should only provide enough detail to make a reasonable low risk estimate of how long the story will take to implement. When the stories shall be implemented, the programmers can contact the customers and receive a detailed description of the requirements orally. User stories have focus on user needs.

IEC 61508: See IEC 61508 specifies documents to be produced during the development. Table 11 shows the objectives for each document.

Table 11 contains description of the objectives of the requirement document according to IEC 61508. The specification of the requirements for software safety shall be sufficiently detailed to allow the design and implementation to achieve the required safety integrity level, and to allow an assessment of functional safety to be carried out (IEC 61508-3 7.2.2.3).

To the extent required by the chosen safety integrity level, the specified requirements for software safety shall be expressed and structured so that they are (IEC 61508-3 7.2.2.6):

- Clear, precise, unequivocal, verifiable, testable, maintainable and feasible, commensurate with the safety integrity level
- Traceable back to the specification of the safety requirements of the E/E/PE safety-related system

- Free of terminology and description which are ambiguous and/or not understood by those who will utilize the document at any stage of the software safety lifecycle.

Comments: The requirement document described in IEC 61508 is more formal than XPs user stories. It is also traceable back to the specification of the safety requirements that is out of the scope of XP. See Chapter 5 for a further discussion of requirement document.

4.7.3.2 Architecture

XP: In XP, part of the architecture is captured by the system metaphor. A good metaphor can tell the team how the system works. The metaphor guides all development with a simple, shared story of how the overall system works.

IEC 61508: A description of the software architecture shall be detailed. See IEC 61508-3 7.4.3.2 for details.

Comments: The system metaphor described in XP is not sufficient for developing a safety-critical system. It is important to have a stable architecture which limits the need for modification. The modification cost in a safety-critical system is considerable higher than in a non safety-critical system. See Chapter 5 for a further discussion of the architecture document.

Design

XP: In XP, the design is generally not explicitly documented; it is described by the source code.

IEC 61508: The design representation shall be based on a notation which is unambiguously defined or restricted to unambiguously defined features (IEC 61508-3 7.4.2.5).

Comments: The different requirement about design document does contradict. When developing a safety-critical system it is necessary to have some documentation of the design. See Chapter 5 for a further discussion.

Coding

XP: The XP programmers write the source code according to a coding standard.

IEC 61508: The source code shall according to IEC 61508-3 7.4.6.1:

- Be readable, understandable and testable
- Satisfy the specified requirements for software module design
- Satisfy the specified requirements for the coding standards
- Satisfy all relevant requirements specified during safety planning

Comments: The source code does not contradict IEC 61508 since the requirements in XP (write code according to the coding standard), does not contradict (see section 4.4.3).

Review
XP: Despite continually code review, there is not produced any review documents.

IEC 61508: Each module of software code should be reviewed. There is no explicit requirement for producing a physical document of the result.

Comments: Since IEC 61508 does not require a physical document of the review result, this process does not contradict IEC 61508.

Testing
XP: Unit test and acceptance test are written in XP.

IEC 61508: The result of software module and integration testing shall be documented, stating the test results, and whether the objectives and criteria of the test criteria have been met. If there is a failure, the reasons for the failure shall be documented (IEC 61508-3 7.4.7.3, 7.4.8.4).

Comments: The unit and acceptance test in XP can be written in such a way or the developer can use special testing tools that make it possible to easily extract the result. If such methods are used, it can be said that XP and IEX 61508 does not contradict with respect to testing results. See Chapter 5 for a further discussion.

Validation
XP: Validation is not in the scope of XP.

IEC 61508: IEC 61508 specifies documents to be produced during the development. Table 11 shows the objectives for the documents.

The plan for validating the software safety shall consider the following (IEC 61508-3 7.3.2.2):

- When the validation shall take place
- Who shall carry out the validation
- The relevant modes of the EUC operation
- Identification of the safety-related software which needs to be validated for each mode of EUC operation before commissioning commences
- The technical strategy for the validation, for example analytic methods, statistic tests etc. (see IEC 61508-3 7.3.2.3)
- The measures and procedure that shall be used for confirming that each safety function comply with the specified requirements for the software safety functions, and the specified requirements for software safety integrity
- Specific reference to the specified requirements for software safety (see IEC 61508-3 7.2)
- The required environment in which the validation activities are to take place
- The pass/fail criteria (see IEC 61508-3 7.3.2.5)

- The policies and procedures for evaluating the results of the validation, particularly failures

Comments: IEC 61508s requirement for validation plan can not be said to be in contradiction to XP because validation is not in the scope of XP. See Chapter 5 for a discussion of the development aspect and safety aspect.

Verification
XP: There is no verification document produced in XP.

IEC 61508: IEC 61508 specifies documents to be produced during the development. Table 11 shows the objectives for the documents.

Comments: See Chapter 5 for a further discussion.

Assessment
XP: XP does not perform any assessment.

IEC 61508: IEC 61508 specifies documents to be produced during the development. Table 11 shows the objectives for the documents.

Comments: See Chapter 5 for a further discussion.

Modification
XP: The source code can be constantly modified by refactoring. Refactoring involves change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behaviour. Since refactoring is done continually through out the development process, the source code is constantly modified. XP does not produce any document which prove modifications.

IEC 61508: IEC 61508 specifies documents to be produced during the development. Table 11 shows the objectives for the documents.

Prior to carrying out any modification activity, the necessary procedures shall be planned (IEC 61508-1 7.16.2.1). The modification and retrofit phase shall be initiated only by the issue of an authorized request under the procedure for the management of functional safety (IEC 61508-1 7.16.2.2). An impact analysis shall be carried out before a modification is performed (see IEC 61508-1 7.16.2.3).

Comments: The modification procedure of XP and IEC 61508 are in contradiction. XP provides no plan what so ever, and IEC 61508 requires a comprehensive description of the procedure. See Chapter 5 for a further discussion.

5 Compatibility of IEC 61508 and XP

"The most striking difference between most 'non-critical' systems and safety-critical systems is that the non-critical systems are only required to work when 'all is well', but safety-critical systems must work even in the presence of failure" [42]. This difference places restrictions to the development process of safety-related systems. The overall safety lifecycle should be used as a basis for claiming conformance to IEC 61508. Nevertheless, a different overall safety lifecycle can be used providing the objectives and requirements in each clause of the standard are met. Chapter 4 pointed out differences and conformity between XP and IEC 61508, and adjustments to minor differences were proposed. In this chapter recommendation to a modified safety lifecycle are presented. These guidelines do not conflict with IEC 61508. Safety in software is expensive to implement. The goal of the proposed model is therefore to achieve an appropriate level of safety with as low cost as possible by using XP practices.

System and software safety plans and tasks should not be separated. Software development includes the process of defining software safety activities [48]. Therefore, XP activities will be integrated into the safety activities. First a modified software safety lifecycle is presented in subchapter 5.1. The following subchapters in this chapter are building on this lifecycle; requirements, the planning game, architecture, design, prototyping, implementation, modification, integration and integration testing. In addition this chapter further discuss release, reliability, verification, validation, assessment, documentation. Finally we present a summary.

5.1 Development cycle

Safety considerations affect all stages of a system's life. It also includes everybody who is working with it: customer, designers, those responsible for implementation and installation, maintenance staff, operator, and users of the system. The safety aspects of the development process can be summarized as follows [42]:

1. Identify potential hazards.
2. Design and verify the system to show that the hazards will not (are sufficiently unlikely to) arise.
3. Analyze possible failure modes and show that safety is maintained even in the presence of failure – the key aspect of validation.
4. Control the development process and produce documentary evidence so that it is manifest that you have done 1, 2, and 3 properly, both in absolute terms and against any relevant standards.

IEC 61508 states the requirements for achieving safety integrity for the software. It requires a combination of fault avoidance (quality assurance) and fault tolerance approaches (software architecture). The standard suggests software engineering principles such as top down design, modularity, verification of each phase of the development lifecycle, verified software modules and software module libraries, and clear documentation to facilitate verification and validation.

In order to comply with IEC 61508, safety activities have to be carried out. XP can therefore only replace part of the safety lifecycle. We will in this chapter discuss how XP can replace or contribute in the execution of some phases of the safety lifecycle. Phase four and five of the safety lifecycle focus on the safety requirements, and phase nine covers the software safety lifecycle (see Figure 8). We will therefore focus on these phases in particular. The other phases in the safety lifecycle focus on other areas than software development, and have in all likelihood to be carried out to comply with IEC 61508.

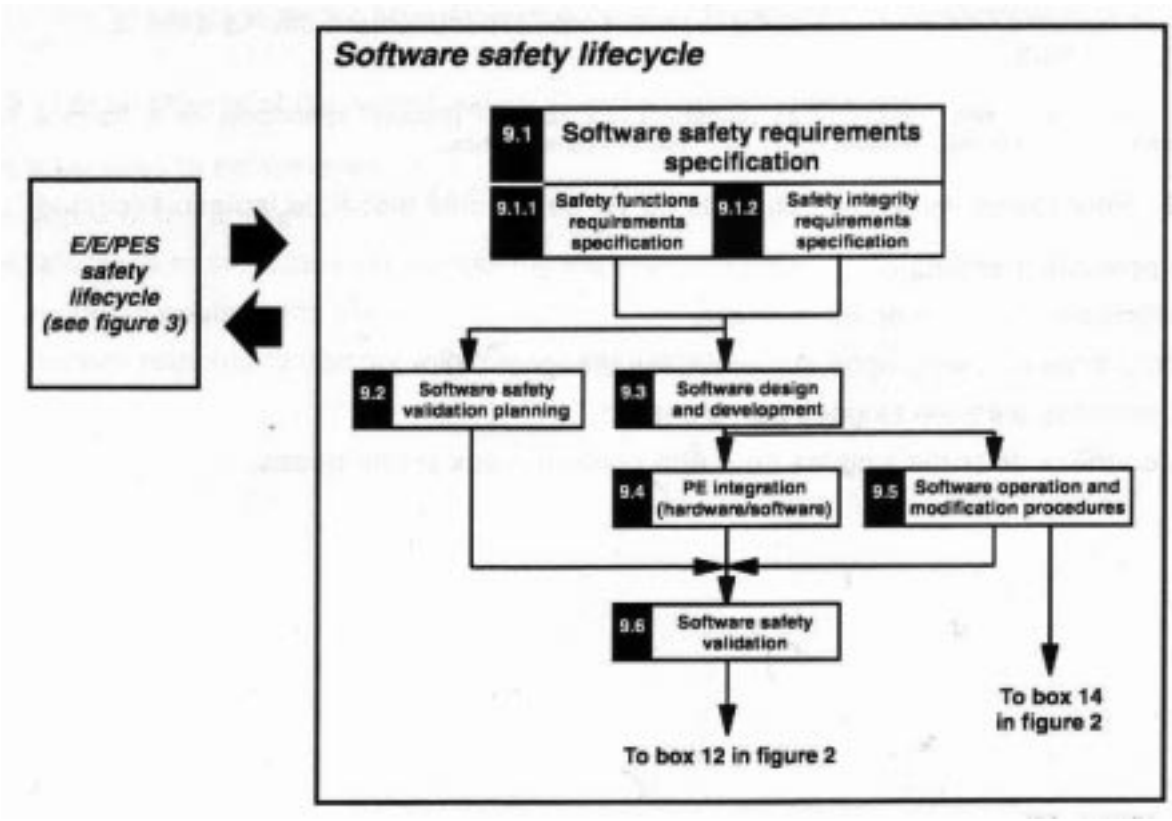


Figure 8 Software safety lifecycle [25]

When developing simple systems, some safety lifecycle phases can be merged. The lifecycle phases are suitable for large, newly developed systems. In small system, it might be appropriate, for example, to merge the phases of software system design and architectural design [25]. As discussed in Chapter 4, the number of developers should not exceed ten persons (according to XP), so the system to be developed is likely to be small. It is therefore recommendable to follow the proposal of merging the phases of software system design and architectural design. See subchapter 5.4 for a discussion of the architecture phase.

Figure 9 shows a modified diagram for the software safety lifecycle which illustrates iterations in the development. Several of the boxes in Figure 8 are also represented in Figure 9; software safety requirements specification, software safety validation planning,

PE integration, and software safety validation. Box 9.3 *software design and development* in Figure 8 is replaced with box 9.4 *software design/development/unit testing* in Figure 9. In the proposed module, unit testing have a more important role than in the software safety lifecycle from IEC. In the same way, box 9.5 *software operation and modification procedures* in Figure 8 is replaced with box 9.5 *software modification/refactoring* in Figure 9. Refactoring is an important part of software development in XP. As a result of this the code is constantly modified to make it more understandable. In the proposed model software integration is extracted from box 9.3 in Figure 8 to illustrate the importance of the integration and the sequence of actions that are performed. Box 9.3 *planning game* is the only new contribution to the model. Since this model supports changing requirements there is a need for planning. The objectives of the planning game are that important stories are understood and estimated. Less important stories can be understood during the development. Every phase in the IEC software safety lifecycle is covered in the proposed model.

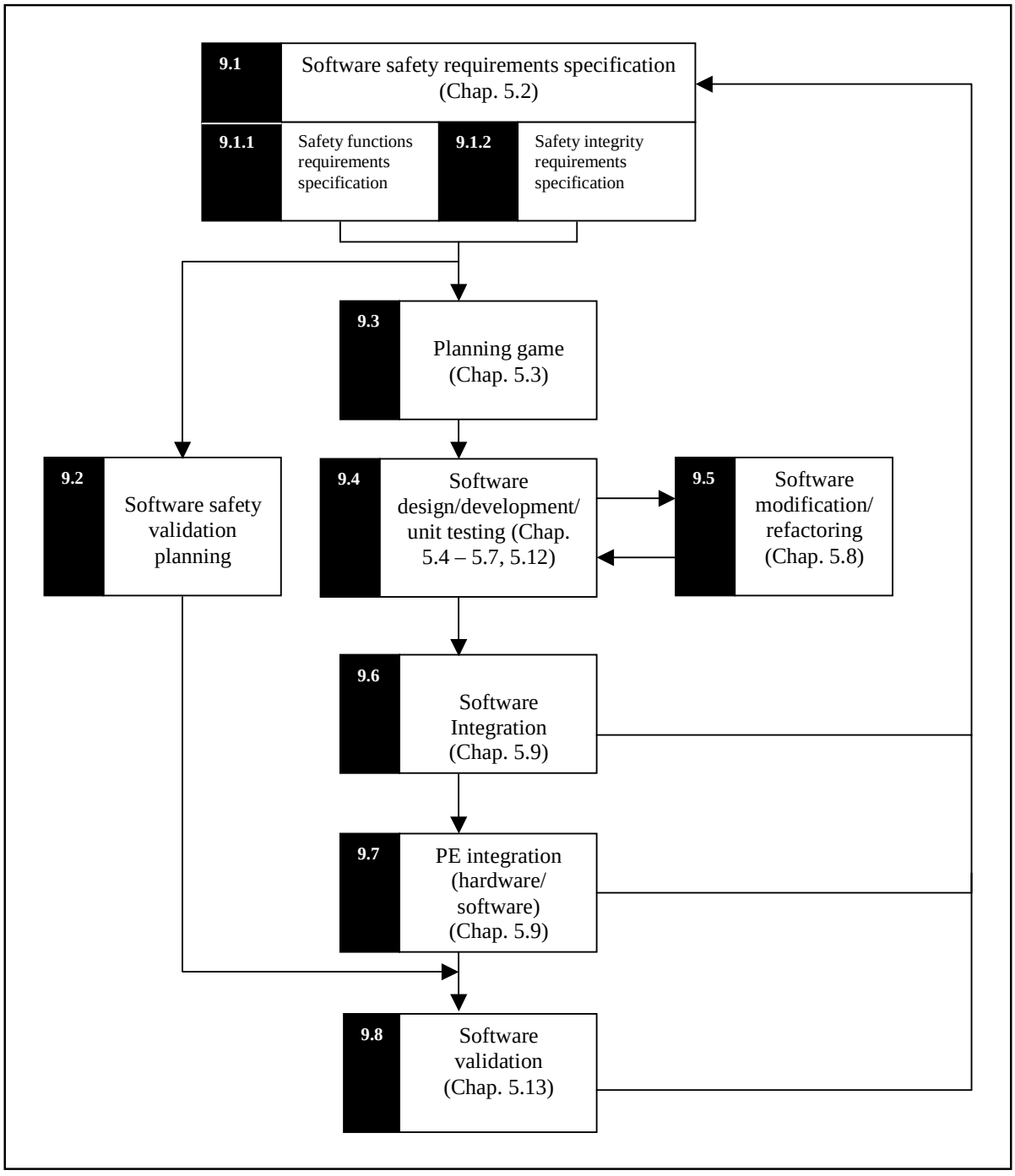


Figure 9 Proposed software safety lifecycle

The rest of the subchapters in this chapter are building on this lifecycle model, and will discuss every phase of the model.

The purpose of Table 13 is to clarify the software safety lifecycle, proposed in Figure 9, and comply with IEC 61508-3 requirement 7.1.2.3. This requirement demands that each phase of the proposed software safety lifecycle shall be divided into elementary activities with the scope, input and output specified.

Table 13 Proposed software safety lifecycle

Safety lifecycle phase		**Objectives**	**Scope**	**Requirements subclause / recom-mendations**	**Input (information required)**	**Outputs (information produced)**
Figure 2 box number	**Title**					
9.1	Software safety requirements specification	To specify the requirements for software safety in terms of the requirements for software safety integrity. To specify the requirements for the software safety functions for each E/E/PE safety-related system necessary to implement the required safety functions. To specify the requirements for software safety integrity for each E/E/PE safety-related system necessary to achieve the safety integrity level specified for each safety function allocated to that E/E/PE safety-related system.	PES, software system	IEC 61508-3 7.2.2	E/E/PES safety requirements specification (IEC 61508-2)	Software safety requirements specification
9.2	Software safety validation planning	To develop a plan for validating the software safety.	PES, software system	IEC 61508-3 7.3.2, see subchapter 5.13.	Software safety requirements specification	Software safety validation plan. Acceptance test specification.
9.3	Planning game	To develop a plan for the programming task. Understand the stories and estimate them.	PES, software system	See subchapter 5.3.	Software safety requirements specification	Software iteration plan.
9.4	Software design/ development/ unit testing	**Support tools and programming languages:** To select a suitable set of tools, including languages and compilers, for the required safety integrity level, over the whole safety lifecycle of the software which assists verification, validation, assessment and modification.	PES, software system, support tools, program-ming language	IEC 61508-3 7.4.4, see Chapter 4 section 4.4.2, 4.4.3, and 4.5.5.	Software safety requirements specification. Software architecture design description.	Development tools and coding standards. Selection of development tools.

Table 1 (continued)

Safety lifecycle phase		Objectives	Scope	Requirements subclause/ recom-mendations	Input (information required)	Outputs (information produced)
Figure 2 box number	**Title**					
9.4	Software design/ development/ unit testing	**Architecture:** To create a software architecture that fulfils the specified requirements for software safety with respect to the required safety integrity level. To review and evaluate the requirements placed on the software by the hardware architecture of the E/E/PE safety-related system, including the significance of E/E/PE hardware/software interactions for safety of the equipment under control.	PES, software system.	IEC 61508-3 7.4.3, see subchapter 5.4.	Software safety requirements specification. E/E/PES hardware architecture design (from IEC 61508-2).	Software architecture design description. Software acceptance test.
9.4	Software design/ development/ unit testing	**Support tools and programming languages:** To select a suitable set of tools, including languages and compilers, for the required safety integrity level, over the whole safety lifecycle of the software which assists verification, validation, assessment and modification.	PES, software system, support tools, program-ming language	IEC 61508-3 7.4.4, see Chapter 4 section 4.4.2, 4.4.3, and 4.5.5.	Software safety requirements specification. Software architecture design description.	Development tools and coding standards. Selection of development tools.
9.4	Software design/ development/ unit testing	**Detailed design and development:** To design and implement software that fulfils the specified requirements for software safety with respect to the required safety integrity level, which is analyzable and verifiable, and which is capable of being safely modified.	Software system design.	IEC 61508-3 7.4.5, see subchapter 5.4.	Software architecture design description. Support tools and coding standard.	Software unit design specification. Unit test specification.
9.4	Software design/ development/ unit testing	**Detailed code implementation:** To design and implement software that fulfils the specified requirements for software safety with respect to the required safety integrity level, which is analyzable and verifiable, and which is capable of being safely modified.	Individual software units.	IEC 61508-3 7.4.6, see subchapter 5.7.	Software unit design specification. Support tools and coding standard.	Source code. Code review.

Table 1 (concluded)

Safety lifecycle phase		Objectives	Scope	Requirements subclause / recom-mendations	Input (information required)	Outputs (information produced)
Figure 2 box number	**Title**					
9.4	Software design/ development/ unit testing	**Software unit testing:** To verify that the requirements for software safety have been achieved – to show that each software unit performs its intended function and does not perform unintended functions.	Software unit.	IEC 61508-3 7.4.7, see subchapter 5.7.	Software unit test specification. Source code.	Software unit test results. Verified and tested software units.
9.5	Software modification/ refactoring	To make corrections, enhancements or adaptations to the software, ensuring that the required software safety integrity level is sustained.	Individual software units.	See subchapter 5.8.	Software unit test specification. Source code.	Software unit test specification. Source code.
9.6	Software integration	To verify that the requirements for software safety have been achieved – to show that all software units, components and subsystems interacts correctly to perform their intended functions and do not perform unintended functions.	Software architecture Software system.	IEC 61508-3 7.4.8, see subchapter 5.9.	Software acceptance test specification. Simulation of real hardware.	Software acceptance test results. Verified and tested software system.
9.7	PE integration (hardware/ software)	To integrate the software onto the target programmable electronic hardware. To combine the software and hardware in the safety-related programmable electronics of the intended safety integrity level.	Program-able electronics hardware. Integrated software.	IEC 61508-3 7.5.2, see subchapter 5.9.	Acceptance test specification. Integrated program-mable electronics.	Acceptance test results. Verified and tested integrated programmable electronics.
9.8	Software safety validation	To ensure that the integrated system complies with the specified requirements for software safety at the intended safety integrity level.	Program-able electronics hardware. Integrated software.	IEC 61508-3 7.7.2	Software safety validation plan.	Software safety validation results. Validated software.
-	Software functional safety assessment	To investigate and arrive at a judgment on the functional safety achieved by the E/E/PE safety-related systems.	All above phases.	IEC 61508-3 8, see subchapter 5.14.	Software functional safety assessment plan.	Software functional safety assessment report.

It should be noted that the creation of an architecture and selection of support tools and programming languages from box 9.4 should only be carried out as part of the first iteration. Preceding iteration can skip these activities, since the result from the first iteration can be used.

According to requirement IEC 61508-3, requirements 7.4.5.2, the following information should be available prior to the start of detailed design: the specification of requirements for software safety (see IEC 61508-3 7.2); the description of the software architecture design (see IEC 61508-3 7.4.3); the plan for validating the software safety (see IEC 61508-3 7.3). By studying Figure 9 and Table 13, we see that this requirement is fulfilled.

5.1.1 Hazard- and risk analysis

Hazard- and risk analysis is phase three in the overall safety lifecycle, and an important step to achieve a safe system. The value of early hazard analysis has been thoroughly discussed by Storey [54] and Leveson [35]. In addition to the requirement presented in IEC 61508 we will give some suggestions to improvements or guidelines to the execution of the analysis.

The hazard- and risk analysis are carried out by a team whose members are chosen by their ability to bring complementary viewpoints to the process. Hazard and risk analysis is mainly a activity done by people. Thus, the quality of the work done by the people involved is a critical success factor. The personnel that shall perform the analysis must be experienced and have knowledge in the areas of system's development, the system's application area, hazard analyses and assessment, and system's operation and maintenance [5]. Normally an XP team does not have these qualities. In these cases personnel outside the XP team should perform the analyses. We recommend that the project manager and the customers, which is a part of the XP team, are present during this process to gain safety knowledge that can be forwarded to the project team.

Misunderstood or incompleted requirements are the source of most operational errors and almost all software contributions to accidents. The problem is dealing with complexity due to lots of complex requirement. One step in controlling complexity is to separate external behavior from complexity of internal design to accomplish the behavior [46].

If following the proposal in this report, the software part of system is developed in iterations (see subchapter 5.1.2). For each iteration an impact analysis will have to be carried out. If the programmers and customer have the right competence and experience to carry out the analysis, they will perform the analysis. Otherwise persons outside the team will do the work. The analysis shall determine if a hazard- and risk analysis should be performed to detect hazards that arise due to new requirements. Hazard analysis and controls is a continuous, iterative process throughout system development and use. Hazard identification should begin as early as the conceptual development of the system and continue all the way through. As soon as design begins, the programmers focus on how the hazards can be controlled. During development, it is possible to verify that hazards have, in fact, been controlled by the design measures already imposed [49].

5.1.2 Iterative development

It has been argued that it is not possible to first define the entire problem, design the entire solution, and then build the software and test the system. An iterative approach seem to be the solution. It allows an increased understanding of the problem through

successive refinement. High-level requirements are defined in phase five in the overall safety lifecycle presented in Chapter 3. If these are to be changed, one should follow the requirements stated in IEC 61508. The high-level requirements can further be decomposed into more detailed requirements. When following our proposal illustrated in Figure 9, one can easily change the decomposed requirement.

Even though the phases in the lifecycles presented in IEC 61508 are read as sequential, they do not necessarily have to be executed sequentially. The overall lifecycle (see Figure 1 in Chapter 3), E/E/PES lifecycle and software lifecycle (see Figure 8) are simplified views of reality, and do not show all the iterations relating to specific phases or between phases. Iteration, however, is an essential and vital part of development through the overall, E/E/PES and software safety lifecycles (IEC 61508-1 7.1.1.4). The safety lifecycle is an approximation. It portrays its phases as being sequential, but several activities are iterative, e.g. specifying the software safety functions and software integrity levels and hazard- and risk analysis. Having this perspective to the safety lifecycle models makes it easier to accept our proposed model. When using XP practices to develop safety-related system, even more iterations will take place during the development. See Figure 9 for an illustration.

The iterations illustrated in Figure 9 follow most of XP practices: User stories to be implemented are chosen in a planning game, and the developers evaluate and accept their responsibilities. Pair-programming can be extensively used. Unit tests are written before the code, and is an important part of the verification. A single iteration can involve further requirement assessment and analysis such as hazard and risk analysis. We recommend keeping the system as simple as possible, emphasizing the quality of the code and how well it fits into the system architecture.

When tasks in an iteration are integrated into the system, new failure modes may be discovered which introduces new event sequence leading to hazardous events. Therefore a new hazard and risk analysis may be necessary.

After the iteration phase is completed, a team consisting of qualified people evaluates the result, and thereafter give feedback to the customer. There will be many opportunities to make correction due to the rapid feedback. It is easier to control the development by making many small adjustments. In addition it lets the developer learn form each increments.

5.2 Requirements

Box 9.1 of Figure 9 *proposed software safety lifecycle* contains the specification of the software safety requirements. This subchapter discusses the activities related to that phase, and requirements in general form.

Most software requirements only specify nominal behavior, i.e. requirements are written to explain what software must and should do. Safety is, however, a property that impacts software by specifying what software must not do. Disallowing anything outside the specification is a demand that in all likelihood is impossible to meet. The system hazard

analysis can derive constraints on the behavior of software. The software component designers and developers can then take these constraints - with traceability maintained from the system-level analyses, and write software that is safe within the context of the system [50].

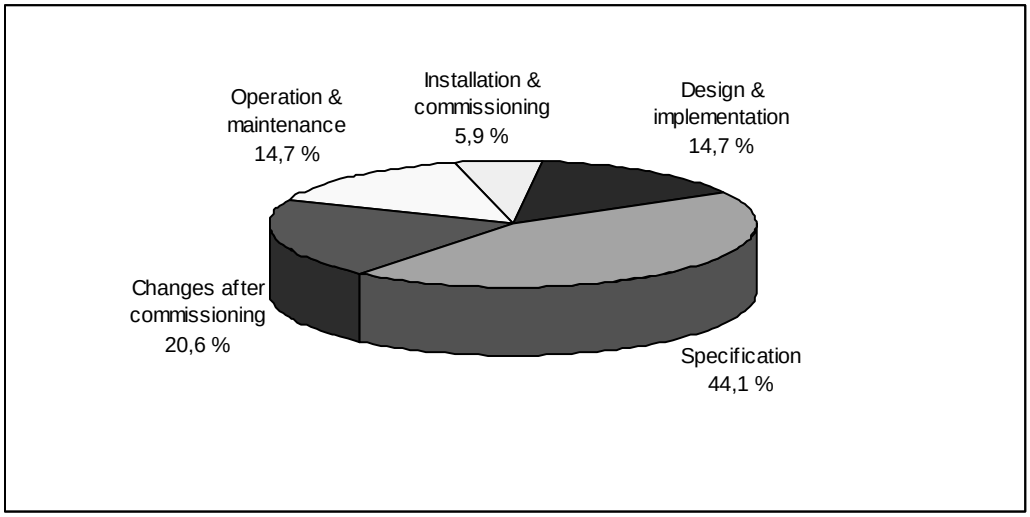


Figure 10 Primary cause of control system failure [5]

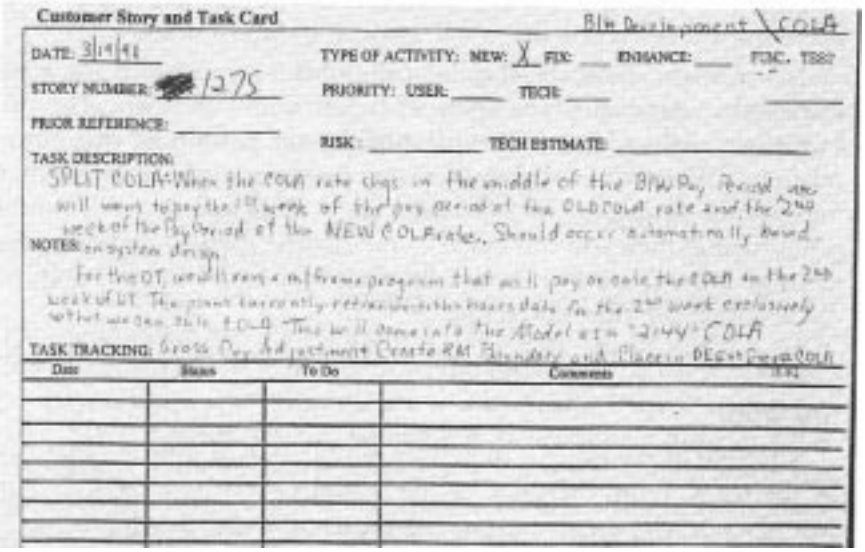
Figure 10 shows, based on 34 incidents, the primary cause (by phase) of control system failure. 44,1 percentage of the failure is caused by specification. It is therefore important to pay extra attention to that phase. Strategies to achieve functional safety for safety-related systems are management of functional safety, technical requirements and competence of persons [5].

Most errors found in operational software can be traced to requirements flaws, particularly incompleteness [34]. It is therefore recommendable to pay extra attention to how requirements are formulated. The requirements have to be specified in an unambiguous way, and some requirements need a formal description depending on the criticality and the impact it has on the system. They should be sufficiently detailed to allow the design and implementation to achieve the required safety integrity, and to allow an assessment of functional safety to be carried out (IEC 61508-3 7.2.2.3). Requirement 7.2.2.6 in IEC 61508-3 states how the requirement for software safety shall be expressed and structured. They should be:

- Clear, precise, unequivocal, verifiable, testable, maintainable and feasible, commensurate with the safety integrity level
- Traceable back to the specification of the safety requirements of the E/E/PE safety-related system
- Free of terminology and descriptions which are ambiguous and/or not understood by those who will utilize the document an any stage of the safety lifecycle.

Despite these strict requirements, most functional requirements will fit into a task card (see Figure 11). If it does not, the card can point to additional documents where it can be

described in detail. If the requirements should be written in an incompleted way, the developers have the possibility to discuss and clear up any uncertainty with the on-cite customer and safety specialists.


Customer Story and Task Card

DATE: 3/19/98

STORY NUMBER: 1275

PRIOR REFERENCE:

TYPE OF ACTIVITY: NEW: X FIX: ENHANCE:

PRIORITY: USER: TECH:

RISK: TECH ESTIMATE:

TASK DESCRIPTION:

NOTES:

TASK TRACKING:

Date	Status	To Do	Comments

Figure 11 A story card [1]

IEC 61508 expects requirements to be specified before designing the system. XP argues that the requirements are never clear at first. The customer does not know exactly what he wants. Therefore, some requirements (user stories) are specified during the development. User stories are made up of two components. The written card is the first. The second component, and by far the most important, is the series of conversations that will take place between the customer and the programmer, viewed in the light of the story. These conversations can clarify any misunderstanding. The high-level requirements will have to be specified in phase five of the overall safety lifecycle. These requirements can be decomposed in many ways. The possibility of new formulation and, to some degree, new requirements, adds the flexibility to the system that XP requires. For instance, one can change the type of hardware device. The assumption of stable high-level requirements makes it possible to define a stable architecture. Since the architecture is based on these requirements. If these requirements do not change the architecture should not change either.

We recommend writing the story in just a couple of sentences on a card *and then pointing to any supporting documentation*. The task cards can be placed on a board according to XP (see Figure 12), and additional comments can be written on the back on the card. All closed tasks are removed from the board. Data, such as tasks risk and task tracking, can be extracted from the card. The task card can further be put into a spreadsheet that makes them easy to trace. When following this procedure, unambiguous requirements are written, and a separate requirement document is produced. When every task is implemented, the requirement document is completed.

Figure 12 Task cards placed on a storyboard [40][40]

In safety-related systems, safety issues also form part of the requirements. In addition, there is a need to communicate requirements outside the team. It is therefore common to produce a separate safety requirements document which sets out what is required of the system to ensure adequate safety. Safety requirements can be specified as user stories independently of functional requirements. These user stories can be placed in a separate document.

The stories must be validated to make sure that they are consistent with each other. When specifying user stories one should ensure that they could be validated independently.

If a task is estimated to take more than a few days, it should be broken down into smaller tasks [1]. In many cases the customer knows what he wants, but does not understand the details of the logic behind it. Technical safety considerations can be outside their expertise. In these cases safety specialist can be helpful. Specialist can decompose comprehensive tasks, in the same way as XP recommend. A story can include element from both software and hardware. The outcome of some events can be dependable of earlier events, and the time elapsed since these events have occurred. The previous course of events can influence the output. It is important that the stories include such information. That information is used to make good tests. The tests in XP are more critical than traditional tests, since the test can be seen as a part of the specification. Features not described in the stories will be missing, and therefore not tested.

To understand the software components of a system, essential information about the components interacting with the environment must be captured. Thus, the user stories should include essential assumptions about the environment in which the software will operate.

Requirements tracing is largely a matter of documentation, with each requirement identified and followed through the implementation and testing process. Since XP's base documentation processes are largely informal, requirements tracing would almost certainly require a substantial increase in the paperwork in the project. One solution is

described above, but to make the traceability even easier, the stories can be associated with acceptance tests.

In XP, the customer's most important requirements is implemented first, so if further functionality has to be dropped it is less important than the functionality that is already running in the system [1]. The safety requirement will often be implemented first because the customer will give them the highest priority.

5.3 The planning game

The goal of the planning game is to determine the next release's or iteration's scope, combining business priorities and technical estimates. The customer decides scope, priority, and dates from a business perspective, whereas technical people estimate and track progress. The cards of the planning reflect the current status of the project. By studying the cards, the team members will know who is responsible for which tasks, what tasks have to be solved next, and what tasks are already solved. The planning game is included in the proposed software safety model in Figure 9.

Once the hazards have been identified, the software safety requirements determined, and the planning for the next iteration is completed, the system must be built to minimize risk and to satisfy these requirements. The next subchapter describes some of the methods that can be used.

5.4 Architecture

From a safety viewpoint, the software architecture phase is where the basic safety strategy is developed for the software. The software architecture defines the major components and subsystems of the software. It identifies how they are interconnected, and how the required attributes, particularly safety integrity, will be achieved. Major software components include systems, databases, plant input/output subsystems, communication subsystems, application program(s), programming and diagnostic tools etc [25]. Software architecture has two objectives according to IEC 61508. The first is to make an architecture that fulfils the specified requirements for software safety with respect to the required safety integrity level. The second objectives is to review and evaluate the requirements placed on the software by the hardware architecture of the E/E/PE safety-related system, including the significance of E/E/PE hardware/software interactions for safety of the equipment under control.

When designing and developing a new system, the system's architecture is one of the variables that can be used to remove hazards and reduce risks. An architecture can be chosen, a hazard and risk analyses can be performed, and it can be checked that the risk is low enough. If it is not, the architecture may need to change and the process repeated **[5]**.

XP specifies the system architecture through a system metaphor. This description will be too vague for a safety-related system. According to IEC 61508-3 requirement 7.4.3.2 the description of the software architecture design shall be detailed. The description shall:

- Select and justify an integrated set of techniques and measures during the software safety lifecycle phases to satisfy the specification of requirements for

software safety at the required safety integrity level. These techniques and measures include software design strategies for both fault tolerance (consistent with the hardware) and fault avoidance, including (where appropriate) redundancy and diversity.

- Be based on a partitioning into components/subsystems, for each of which the following information shall be provided:
  - Whether they are new, existing or proprietary
  - Whether they have been previously verified, and if yes, their verification conditions
  - Whether each subsystem/component is safety-related or not
  - The software safety integrity level of the subsystem/component
- Determine all software/hardware interactions and evaluate and detail their significance.
- Use a notation to represent the architecture which is unambiguously defined or restricted to unambiguously defined features.
- Select the design features to be used for maintaining the safety integrity of all data. Such data may include plant input-output data, communications data, operator interface data, maintenance data and internal database data.
- Specify appropriate software architecture integration tests to ensure that the software architecture satisfies the specification of requirements for software safety at the required safety integrity level.

In order to be able to verify the architectural design process, to ensure that the resulting top-level design is a true representation of the specification, the architecture specification have to be detailed. The architecture should point out the features that influence each module. This identification can ease the integration of new requirements. An early, stable architecture is important to avoid costly modification or changes of it later in the development process. Therefore, the creation of the architecture will only be performed during the first iteration illustrated in Figure 9. As discussed earlier, the assumption of stable high-level requirements from phase five of the overall safety lifecycle, makes it possible to define a stable architecture. To conform to IEC 61508, the requirements described above should be met.

5.5 Design

From the discussion in Chapter 4 we saw that the design practices in XP and IEC 61508 are fundamentally different. Some aspects are nevertheless the same. First, we will focus on those that are equal, and then try to bring closer to each other the different views on design.

Two approaches to achieve safety are simplicity and structure [42]. Simplification may in most situations eliminate hazards. The more complex the design, the more likely it is that errors will be introduced by the protection facilities themselves [47]. Complexity can also be a direct barrier to understanding. A “simple” system has a small number of unknowns in its interactions within the system and with its environment [49]. When the system is simple it is easy to structure. When the structure is good, the system can be divided into parts which are intelligible by themselves. Safeware Engineering Corporation says that

software should only contain code that is absolutely necessary to achieve the required functionality [47]. XP practices the same principles; the design should be as simple as possible, and the design should be organized so that a change in the system affects only a few part of the system. Unit tests in XP also encourage the developers to write many small methods, each responsible for a clear and testable story. In this area IEC 61508 and XP follows the same design rules.

One means of coping with complexity is analytic reduction. The system is divided into distinct parts for analysis purposes, and the parts are examined separately. Analytic reduction relies on three important assumptions [49]:

1. The division into parts will not distort the phenomenon being studied.
2. Components are the same examined singly as when playing their part in the whole.
3. Principles governing the assembling of the components into the whole are themselves straightforward.

Analytic reduction is used when developing safety-related system, but it is also a method used in XP. The method is not mentioned in XP literature, but the unit testing relies strongly on these assumptions.

XP's design philosophy is minimalistic and pragmatic. In contrast to the process used to develop safety-critical system, it does not start with a full up-front analysis and design. A quick analysis of the entire system is carried out, and then the first iteration starts. Grenning [17], when developing safety-related system, discovered that evolutionary design could relieve a lot of pressure from the team. They did not have to create the best design for all time, the design could evolve with the system. These observations support the belief that evolutionary design is possible when developing safety-related system.

In XP the system is constantly modified, in contrast to IEC 61508 where modification are avoided. But IEC 61508 consider the capacity for safe modification during the design activities in order to facilitate implementation of these properties in the final safety-related system (IEC 61508-3 7.4.2.3). Designing for upgrade can significantly reduce the safety and cost impact of future changes. The following should be considered when designing for upgrade: [15]

- Architectural design that includes ease of changes should be considered (e.g. design simplicity, modularity, high cohesion, and low coupling).
- Automated tools for design and verification should be implemented.
- Methods to identify and isolate the impact of an upgrade should be explored (e.g. keeping design data and traceability current).
- Tools to identify the dependencies among different system components should be used (e.g. system modeling tools used to evaluate the dependencies of system components and change to software).
- Cost models should be used to anticipate changes and to evaluate their impact.

XP practices some of these recommendations; ease of change, automated tool for verification.

The design strategy in XP is always to have the simplest design that runs the current test suite. Practicing XP does not include documenting design - “the source code is the design”. Some prefer using CRC-cards to document the design. The level of system representation provided by CRC-cards lies somewhere between unstructured notes and sketches and more formal design techniques. Some of its advantages [8] are easy searching and editing, and it is possible to maintain version control which is important both in software evolution and when developing safety-related system. IEC 61508-3 requirement 7.4.2.5 states that the design representations shall be based on a notation which is unambiguously defined or restricted to unambiguously defined features.

UML Use cases can be used instead of CRC-cards. Johannessen, Grante, Alminger, Eklund, Torin [31] and Grennig [17] have found them useful. Johannessen, Grante, Alminger, Eklund, and Torin integrated a modified Functional Hazard Assessment and Use cases. The analysis generated valuable results used as design requirements and dependability analysis input. They found that their structured way of finding possible functional failures gave almost all the possible functional failures. The UML use case [13] is a tool to capture requirement in early design phases. It is a set of scenarios tied together by common user goal. You do not need to draw a diagram to use *use cases*, it is possible to keep each use case on an index card and sorting the cards into piles to show what needs to be built in each iteration. Grenning [17] did not draw diagram, because he did not think they could add any value to the development team. The similarity between use cases and CRC-cards can imply that CRC-cards also can be used when developing safety-related system.

“The requirement that different design techniques should be used for different integrity levels implies that some methods are demonstrably better than others. There is no scientific or quantitative basis for this assumption” [42]. Still we recommend using CRC-cards or use cases only for SIL1 and SIL2.

Normally, XP does not focus on testability and the capacity for safe modification which is a requirement in IEC 61508-3 (7.4.2.3), but when developing safety-related system this must be taken it into consideration. The safety-related part of the software shall as far as practicable be minimized according to IEC 61508-3 7.4.2.6.

A safe software design includes not only standard software engineering and fault tolerance techniques to enhance reliability, but also special safety features. The next sections discuss design features directly related to safety.

One important aspect of developing safety-related systems is to identify and resolve hazards. The design process is highly affected on the hazard analysis. There is a clear precedence to resolving hazards [50]:

1. Eliminate the hazard
2. Prevent or minimize the occurrence of the hazard
3. Control the hazard if it occurs
4. Minimize the damage

The best alternative is to eliminate the hazard. Often, if done during the requirements specification or design phase, hazard elimination adds no cost. There may be some hazards that can not be eliminated from the system. The goal in this case is to minimize the occurrence of the hazard. One way to do this is to carefully control the conditions under which the system can move from a safe state to a hazardous state. The third option is to try to control the hazard if it occurs. One way to do this is through operator training. These four steps do not have to be taken one at a time. If a hazard can not be eliminated it is still a good idea to try to control the hazard and minimize the damage it may cause if it leads to an accident [50].

Once the potential causes of hazards have been uncovered, design constraints can be placed on the system, software, and human operators. The system can, with these constraints in mind, be designed to eliminate or control hazards. Any hazards that can not be fully resolved within the system-level design must be traced down to component requirements, such as software requirements. This traceability is important, as it is the only way to ensure that remaining hazards are eliminated or controlled within the context of individual components [50].

Designing the system for controllability can reduce hazards. The system can be made easier to control, both for humans and computers. Incremental control can be used. Steps can be performed incrementally rather than in one big step, and provide feedback to test the validity of assumptions and models upon which decisions are made. Feedback may also be provided in terms of intermediate states and partial results. Controllability can be enhanced by lowering time pressures, and perhaps by slowing the process rate [47].

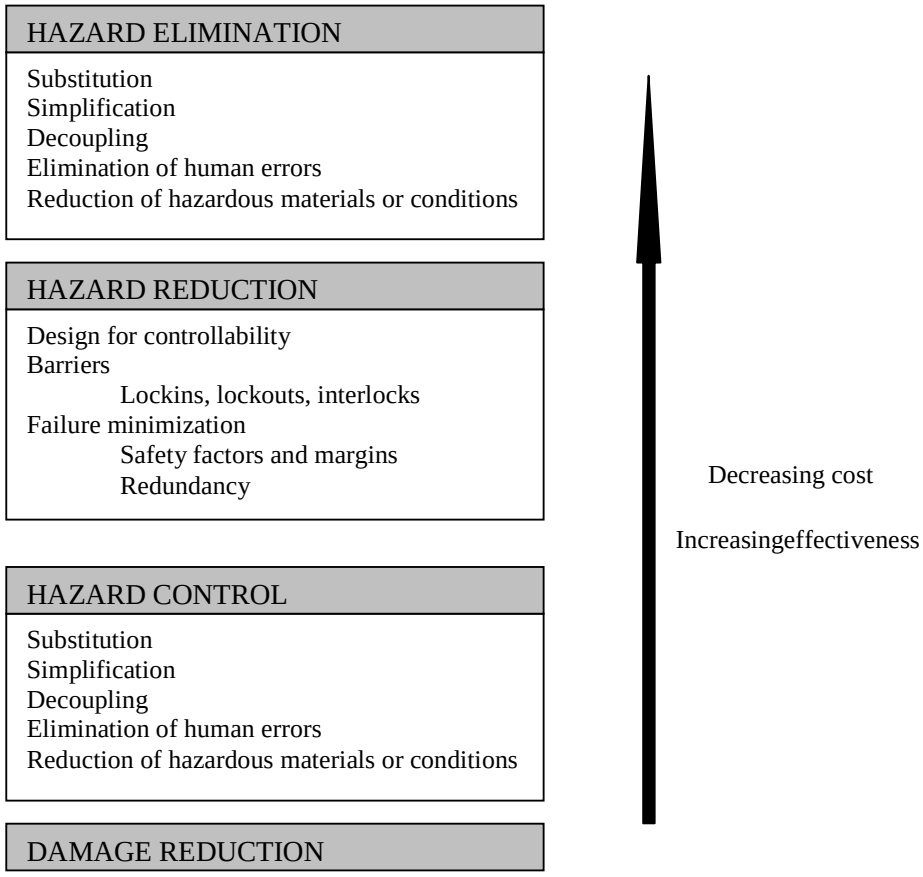


Figure 13 Safe design precedence [47]

5.6 Prototyping

The capability to dynamically analyze, or execute, the description of a software system early in the development lifecycle has many advantages. Dynamic analysis can help the customer to evaluate and address poorly understood aspects of a design, improve communication between people involved in development, allow empirical evaluation of design alternatives, and is one of the more feasible ways of validating a system [57].

When considering a possible prototyping method for safety critical systems the following criteria must be met [57]:

- The language should support methods to assure the correctness of the system
- A prototype of the system should be available early in the development life cycle.

If a subset of C++ is used as the programming language, both of these requirement are fulfilled when following XP. Prototyping can even be used in parallel with phase one to five in the overall safety model. Prototyping/animation is recommended for all SILs in the context of modeling.

One argument against evolutionary prototyping is that it often leads to unstructured code and difficulties with maintaining the systems. Furthermore, incremental changes to the prototype may not be captured in the requirements specification and design documentation which leads to inconsistent documentation and a maintenance nightmare [57]. If the code is constantly refactored, it will be easy to modify and it will be structured. XPs documentation techniques, and the suggestions in subchapter 5.15 do not lead to inconsistent documentation.

5.7 Implementation

After a design session the programmer shall implement software that fulfils the specified requirements for software safety with respect to the safety integrity level. The design, implementation and unit testing are closely connected in our proposal, therefore these activities are placed in the same box in Figure 9. Next follows recommendations how to carry out the implementation. It is recommendable to have the project manager and one or more safety specialists sitting full-time with the development team. They have knowledge from the hazard- and risk analysis and can help the development team explain the system requirements with respect to safety.

Implementation must proceed with safety in mind. Developers should use defensive programming practices. It should not be assumed that correct parameters are passed in or that called functions operate correctly. It is good practice to separate critical functions from the rest of the code. The critical functions can then be more carefully reviewed and audited independently. A clear separation reduces the effort for testing the safety-related system. The interaction between software components should be limited and straightforward. Reducing and simplifying interfaces will eliminate errors and make designs more testable. The safety-critical routines should be kept as small and as simple as possible. Adequate isolation should be achieved between the software modules. These recommendations correspond to IEC 61508-3 where requirements 7.4.5.3 says that the software should be produced to achieve modularity, testability, and capacity for safe modification. In addition, these recommendation do not conflict to XPs programming practices.

IEC 61508-3 has a requirement which includes several guidelines to the source code. The source code shall according to requirement 7.4.6.1:

- Be readable, understandable and testable
- Satisfy the specified requirements for software module design (see IEC 61508-3 7.4.5)
- Satisfy the specified requirements of the coding standards (see IEC 61508-3 7.4.4)
- Satisfy all relevant requirements specified during safety planning (see IEC 61508-3 clause 6)

The three first requirements are also stated in XP. XP does not contain safety planning, but it can correspond to the planning game, and is therefore not in conflict with XP.

Alistair Cockburn and Laurie Williams found that pair programming increased the development expense with 15 %. This initial increase is recovered in the improvements in design quality, reduced defects, reduced staffing risk, enhanced technical skills, improved team communication and it is considered more enjoyable at statistically significant levels [9]. Other publications [60] [61] [39] have also demonstrated that pair programming is beneficial. When the code is continually under code review, mistakes are found as they are entered. This leads to saving the cost of compilation, and providing the economic benefit of early defect identification and removal. Coding standards are followed more accurately with the peer pressure to do so. In addition, team members learn to talk together and work together. As a result of frequently changing partners, system knowledge is shared between the members of the team. Pair programming does not conflict with any of the requirements in IEC 61508, and we believe that pair programming will improve the implementation process of safety-related system. The benefits reported when developing 'non-critical' system will also apply to safety-related system.

5.8 Modification

Kent Beck notes that "maintenance is really the normal state of an XP project" [1]. The programmer who "owns" a task will try to find a partner who is familiar with the code affected to save time. It is therefore likely that at least one of the pair programmer have knowledge of the code to be changed. Thus, the XP programmer will rarely have to understand the code all alone. IEC 61508, on the other hand, have strict requirements when performing a modification, and does not recommend any modification if it is not completely necessary. Considering modifications in a safety-related system is a heavy process which has lead to constraining development and integration procedures. Prior to carrying out any software modification, software modification procedures shall be made available according to IEC 61508-3 requirement 7.8.2.1. Before any modification is performed an impact analysis shall be carried out. This analysis shall determine all software modules impacted, and the necessary re-verification and re-design activities performed in compliance with IEC 61508-3 requirement 7.4.8.5. The analysis shall also determine whether or not a hazard and risk analysis is required, and which software safety lifecycle phases we will need to repeat - IEC 61508-3 7.8.2.3. The impact analysis results obtained shall be documented - IEC 61508-3 7.8.2.4.

"A comprehensive set of unit tests reduces the comprehension space when modifying source code" [11]. After a change is made to the code, the result from running the unit tests will tell whether the change leads to errors. This can reduce the risk and complexity of conducting an impact analysis.

A modification shall, according to IEC 61508, be initiated only on the issue of an authorized software modification request under the procedures specified during safety planning (see clause 6) which details the following (IEC 61508-3 7.8.2.2)

- The hazards which may be affected
- The proposed change
- The reasons for change

The documentation due to a modification is comprehensive when following IEC 61508. Details of all modifications shall be documented, including references to (IEC 61508-3 7.8.2.8):

- The modification/retrofit request
- The results of the impact analysis, which assesses the impact of the proposed software modification on the functional safety, and the decisions taken with associated justifications
- Software configuration management history
- Deviation from normal operations and conditions
- All documented information affected by the modification activity. The documentation shall include (IEC 61508-2 7.8.2.1):
  - The detailed specification of the modification or change
  - An analysis of the impact of the modification activity an the overall system, including hardware, software, human interaction and the environment and possible interactions
  - All approvals for changes
  - Progress of changes
  - Test cases for components including revalidation data
  - E/E/PES configuration management history
  - Deviation from normal operation and conditions
  - Necessary changes to system procedures
  - Necessary changes to documentation.

If all these procedures have to be performed for every refactoring, refactoring will be pointless. A unit test has to be written before any refactoring, and this can to some degree detect most of the failures that can arise due to refactoring. The modification procedures described in IEC 61508 are meant for modification after the software design and development are finished, in contrast to refactoring that is performed during the development. Refactoring and adding new code are compatible. Before new code is added, it is recommended to investigate whether a new hazard and risk analysis should be performed. Refactoring is often carried out before adding new code, and the affect of this process can be taking into consideration in the hazard and risk analysis, and therefore no extra work has to be done. The primary goal of refactoring is to make the code more understandable. Most of refactorings proposed by Fowler [12] leads to a simpler code, which is especially important in safety-related system. Software modification and refactoring is represented in the proposed software safety lifecycle (Figure 9) as box 9.5. The figure illustrates that modifcation and refactiong are performed in parallel with development.

In Chapter 4 we recommend to use C++ as programming language. Using refactoring with C++ is hard [55]. Manfred Lange suggests in addition to Martin Fowler some refactorings that has focus on C++ [55]. He argues that with good tools it is possible to refactor C++ code.

It is difficult to determine the correctness of each change and to perform a through change impact analysis, if changes are made at the same time. In XP, current changes are

part of the development practice. However, software changes can be implemented simultaneously (provided they do not affect the same area of the code), as long as each software change is incorporated individually into an existing baseline [43]. To assure correctness of each change, the change should be verified both individually and as a part of the overall system. Unit tests and acceptance test satisfies this requirement.

Rapid changes in code add risk. We need to be sure of two things: that the new capability works, and that we have not broken anything that used to work. XP makes it possible to develop modifiable code [1]:

- A simple design, with no extra design elements.
- Automated tests, which detects accidentally fault because of change in the existing behavior of the system. These tests are run in almost zero time, and this makes it possible to run all test before and after any change.
- Lots of practice in modifying the design, which leads to confidence to change the system if needed.

If the system have a stable architecture, as discussed in subchapter 5.4, it can limit the need for modification. Small changes can have serious impact, and proper pre-coding planning could help to solve this. We will most likely have fewer modification that in a traditionally XP project.

A software modification can be handled as a minidevelopment [43]. Plans should be established for development, software quality assurance, software configuration management, and verification/testing. Once the plans are established, the changes should be implemented following those plans. It is important to have a defined, structured, and rigorous process, when changing software in safety-critical systems. A process change should include, as a minimum, change impact analysis, change planning, change implementation and verification, problem reporting and analysis, a process to access and address problems, software quality assurance, and software configuration management [43].

5.9 Integration and integration testing

Integration in XP involves merging the changes made with the code written by other developers. Whenever a task card is solved, the solution should be integrated. Integration phases occur several times a day. XP describes only software integration, integration of software and hardware is not within the scope of XP. We have separated software integration and PE integration as different phases in the proposed software safety lifecycle (Figure 9). This separation is done due to the different frequency and activities involved. Software integration will be carried out every time a task is solved. Software integration immediately follows development, including integration testing.

Programmable electronic (PE) integration (box 9.7 in Figure 9) involves combining the software with the programmable electronics “to ensure their compatibility and to meet the requirements of the intended safety integrity level”. Functional, black box and performance testing are suggested as appropriate methods for achieving PE integration. We recommend to use functional and black box testing as described below.

Initial versions of the system can be evaluated in a simplified environment containing e.g. failure free sensors and actuators. Interfaces for the external components can be defined so that failure signals are an input to the software. Failure will therefore not be detected at this stage because the software gets information from a non-realistic, simplified version of the "environment". This allows the programmer to focus on how the software should respond to error conditions rather than how these conditions are detected.

As the understanding of the system and its environment deepens, more user stories are written, and the environmental models will be refined. Thompson and Heimdahl [57], and Grenning [17] proposed a similar approach. Thompson and Heimdahl presented an approach to requirements specification and evaluation that integrates formal requirements specification and rapid prototyping. Alternatively, the hardware part of the system is finished before developing the software part. Software versions of the system can be integrated into the finished hardware part from the beginning of the software development. Grenning identified, during the iteration planning, the interfaces needed to support the iteration features. These acted as placeholders for the real hardware. A simulation of the interface were added which facilitated development and kept volatile entities from the application logic. This made it possible to simulate the systems interactions with its environment.

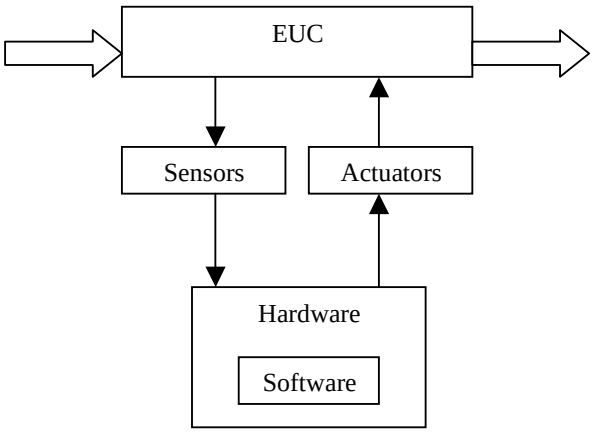


Figure 14 A computer-based control or protection system [54]

The equipment under control (EUC) is the system or equipment with which the application is concerned. This equipment will have input from, and outputs to, the environment. The control or protection system interacts with the EUC through sensors and actuators that are used to monitor and control certain parameters. The logic inside the dashed box, in Figure 15, can be tested. One should test all combinations of input signals. One can easily generate tests for digital signals, since the logic does not have memory. All input are tested to prove that the right output is produced. It is recommended that the interface to the OS be analyzed.

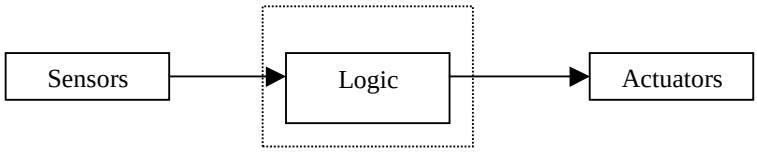


Figure 15 Software integration

Distributed system are harder to test, because the complex composition of input signals and output signals.

According to requirement IEC 61508-3 7.4.5.5 appropriate software system integration tests should be specified to ensure that the software system satisfies the specified requirements for software safety at the required safety integrity level (see 7.2). These tests shall be specified concurrently during design and development phase according to IEC 61508-3 requirement 7.4.8.1. The tests shall specify the following – IEC 61508-3 7.4.8.2:

- The division of the software into manageable integration sets
- Test cases and test data
- Types of tests to be performed
- Test environment, tools, configuration and programs
- Test criteria on which the completion of the test will be judged
- Procedures for corrective action on failure of test

The integration tests for programmable electronics (hardware and software) shall specify the following – IEC 61508 7.5.2.2:

- The split of the system into integration levels
- Test cases and test data
- Types of tests to be performed
- Test environment including tools, support software and configuration description
- Test criteria on which the completion of the test will be judged

The objective of software integration testing, phase 9.3 in safety software lifecycle, is to verify that the requirements for software safety (in terms of the required software safety functions and the software safety integrity) have been achieved. In addition it shall show that all software modules, components and subsystems interact correctly to perform their intended function and do not perform unintended functions. The unit tests and acceptance test shall reveal unintended behavior.

During the integration testing of the safety-related programmable electronics (hardware and software), any modification or change to the integrated system shall be subject, according to IEC 61508-3 7.5.2.6, to an impact analysis which shall determine all software module impacted, and the necessary re-verification activities. This requirement does not conflict with the recommendation described earlier in this chapter.

5.10 Release

The number of releases depends on whether the hardware is finished and available to the team when the development of the software starts. If the hardware is developed in parallel with the software, a release can, at the earliest, take place when the hardware is finished. In any case there will be fewer releases than XP recommends. That is due to the hardware and software integration, in addition to the validation process.

5.11 Reliability

Software reliability ensures that the software performs its specified functions with a realistic, acceptable failure rate. XP provides good reliability, here are some reasons why [29]:

1. Unit tests cover "everything that could possibly break".
2. Acceptance test, independently defined by the customer, test all the requirements.
3. Whenever defects slip through the unit tests, to be detected by the acceptance test, we recommend that the programmers upgrade the unit tests, not only to show the existing defects, but to upgrade the testing practices in general based on what was learned about the "missing" tests.
4. Whenever defects slip through the acceptance tests and are caught by users, the same practice is used to upgrade both acceptance tests and unit tests, and the testing practices.
5. All production code is programmed by two programmers working together. This provides inspection of the code by at least one other person.
6. In XP, code is owned by the team, not by individuals. This mean that over the course of the project, essentially all the code is viewed and edited by even more programmers than the original pair who wrote it. This provides even higher levels of inspection.
7. XP teams release software to users frequently, ideally every couple of weeks. This ensures that the software gets plenty of assessment in the real working environment. This enables the team to build an excellent sense of system quality.

Jeffries [29] thinks that high-reliability software is consistent with XP: "It's my understanding that for high-reliability software, the practices resorted to are very comprehensive testing, and very intensive inspection, with occasional use of proof. All of these practices are consistent with XP and could be added to an XP project without much difficulty."

Reliability and safety have some similarities, but also differences. Reliability and correctness are necessary precursors to safety. High reliability is normally a necessary, but not sufficient, condition to guarantee safety. "Engineering safety does not imply the need to achieve correctness. A system may be incorrect, and unreliable, but safe because its failure modes are not hazardous. Similarly a system may be correct, but unsafe, due to errors in the specification" [42].

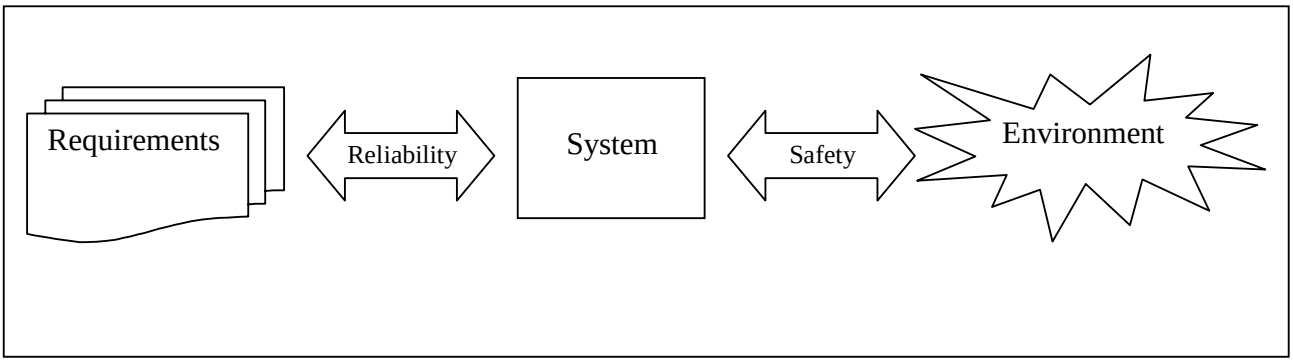


Figure 16 The systems connection to reliability and safety

The reliability of the system is determined by the degree to which it performs its required function. Safety is concerned with the consequences rather than the possibility of a failure. Safety is thus a relation between a system and its surroundings.

Highly reliable components are not necessary safe [49].

1. If the requirements are unsafe from a system perspective, then even correctly implemented software is still unsafe.
2. If the requirements do not specify some particular behavior required for system safety (i.e., the requirements are incomplete), then the software will be unsafe.
3. The requirements can be safe and the software implements those requirements, but the software also has unintended and unsafe behavior beyond what is specified in the requirements.

5.12 Verification

In subchapter 5.9 we discussed integration testing. This subchapter will discuss other verification activities such as unit test, acceptance tests, and code review.

The unit tests and acceptance tests in XP verifies that the software performs functions specified by the customer. The results from the tests show whether the implementation satisfies the specification. Figure 17 illustrates this relationship. The drawback of this approach is that the software can do more than what is specified in the requirements. When verifying a safety-related system this drawback can be crucial.

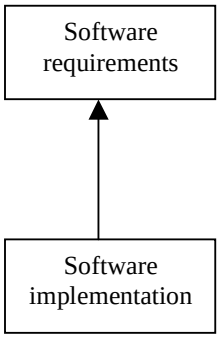


Figure 17 XPs approach to verification

The goal of verifying a safety-related system is to show that the software will not do anything that will lead to violating its constraints. Therefore, the verification of the implementation shall be directed against the safety requirements and constraints as illustrated in Figure 18.

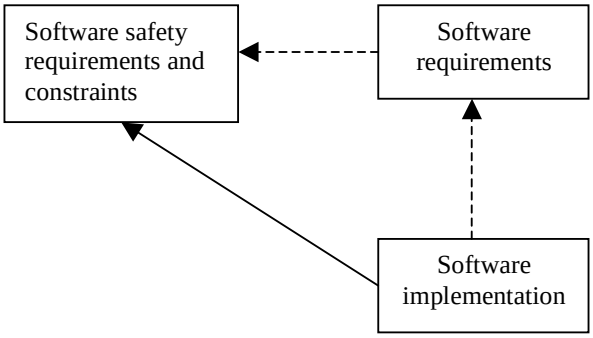


Figure 18 Approach to verifying safety

When developing a safety-critical system, testing is used not only to locate faults within the software, but also as part of the assessment process to gain certification. Static analysis is recommended at SIL 1, and highly recommended at higher levels. IEC 61508 recommend several analysis techniques, but the most important is walkthroughs/design reviews since this is the only technique that is highly recommended for all SIL (see Table 6 in Chapter 4). Continuous design and code review is one of the benefits of pair programming. This review process is probably more comprehensive than what IEC 61508 expect since it is performed both during design and implementation.

Dynamic testing is essential when developing software system. In XP, they normally use unit tests and acceptance tests. These test have to be more comprehensive than in traditionally software development. They should include test cases that check the system for unwanted behavior. The test shall show that each software module performs its intended function and does not perform unintended functions. It is also necessary to execute the testing within the target environment. The unit and acceptance tests in XP are

automatic and can be an improvement, compared to module and integration tests traditionally used when testing safety-related system. It is bothersome to have comprehensive repeatable tests if the result has to be checked manually. The testing framework proposed by Kent Beck [3], checks the results and reports them to the user of the framework. In addition the unit test are done continually, instead of only late in the development. If they break, they have to be fixed immediately. Thus, less reporting has to be done since the test should always be 100%. Several articles [17] [14] [32] have reported significant improvements when using unit tests. Unit test and acceptance tests may be a part of the process to gain certification. Both IEC 61508 and XP suggest performing other test if needed.

The result of the tests shall be documented according to IEC 61508-3 requirement 7.4.8.4. When executing a test by one of the testing framework [3], e.g. CppUnit, the result from the test is displayed on the screen and can easily be copied and used as documentation (according to IEC 61508-3 7.4.7.3 and 7.4.8.4). When the programmer writes a test, he also writes the feedback that will be displayed if the test fails. The feedback should include the reason for the failure according to IEC 61508-3 requirement 7.4.8.4. It is benefitional to have good traceability between code and test, since each test belongs to a piece of code.

In XP the customer, together with a tester from the XP team, should write the acceptance test. When developing safety-related systems, the people that write the test should have the right competence and experience. If the customer and tester do not have these skills, other people should write the tests. In XP, the tests are specified during the design and implementation process. This is also what IEC 61508 requires according to IEC 61508-3 requirements 7.5.2.1.

In practice, the biggest problem with testing is dealing with multiple, correlated failures. The practice of developing safety-critical systems is such that few systems fail due to single failures [42]. Errors are more likely to be found in the interaction between software components than in the design of the individual components [34]. “The system is considered as a whole, not just as a collection of components. System safety takes a larger view of hazards than just components failures. Most accidents come from the complexity of interactions among system components, not from component failures. Because reliability improvements address component failures, safety must go beyond reliability improvement. System safety emphasizes hazard analysis and designing to eliminate or control hazards. The analyses of system safety tend to be qualitative rather than quantitative” [50]. The unit test will possibly not catch these failures, and that is why the acceptance test in XP should be comprehensive.

XP does not specify procedures for corrective action on failure of a test, which is a requirement 7.4.7.4 in IEC 61508-3. In order to comply with the standard these procedures should be written.

XP assumes that pair programming replaces explicit code review. Deursen [11] questions whether the several reported benefits of code review are still applicable if it is done “all

the time". Code review is a short, intense, organized session, specifically, focused on finding and removing errors. Pair programming is a dialog between two people trying to simultaneously program and understand together how to program better [1]. Therefore, XP's decisions to refrain from explicit code reviews is a potential risk: it is, for example, possible that pair programming with separate explicit review sessions yield better results, or can be used to find different types of problems [11]. We believe though, that pair programming can replace code review.

5.13 Validation

Validation process is needed to [5]:

- Proof to customers that the product is applicable for intended purpose.
- Proof to authorities that the product is safe and reliable enough for intended purpose.
- Proof to the manufacturer that the product is ready for the market.
- Have documentation to help future alterations of the product.

The acceptance test can be seen as a validation activity, since it is the customer's responsibility to write these tests. The test specification, acceptance criteria and test results form an essential part of evidence of safety. These validation activities does not fulfill the validation requirements for a safety-related system, and we recommend following the validation planning (IEC 61508-3 subclause 7.8) and validation phase (IEC 61508-3 subclause 7.14) described in IEC 61508. These activities are also presented in Figure 9 which shows the proposal of a software safety lifecycle.

5.14 Assessment

Functional safety assessment is the critical activity that ensures that functional safety has actually been achieved. People carrying out the functional safety assessment should be competent and have adequate independence. They should use these qualities to consider the activities carried out and outputs obtained during each phase of every lifecycle. In addition they should judge the extent to which the objectives and requirements of IEC 61508 have been met. According to IEC 61508-3 requirements 8.2.3, the functional safety assessment shall be applied to all phases throughout the overall, E/E/PES and software safety lifecycles. Those carrying out the functional safety assessment shall consider the activities carried out and the outputs obtained during each phase of the overall, E/E/PES and software safety lifecycles and judge the extent to which the objectives and requirements in IEC 61508 have been met.

We recommend following the assessment requirements described in IEC 61508. The assessment shall begin with the formulation of an assessment plan, detailing the scope of assessment and its basis. The customer and safety specialist shall provide all the evidence required to demonstrate compliance with the criteria.

5.15 Documentation

Documentation is an important activity according to IEC 61508. Requirement 7.1.2.7 in IEC 61508-3 demand that the results of the activities in the software safety lifecycle are

documented. The documentation requirements in IEC 61508 are concerned with information rather than physical documents. The standard declare explicitly when the information shall be contained in a physical document. These documents must include the information necessary for effective execution of subsequent phase and for verification activities according to IEC 61508-1 requirement 5.2.1. In the same way, the documentation should include sufficient information required for the implementation of functional assessment – IEC 61508-1 requirement 5.2.3.

For each phase of the overall safety lifecycle, E/E/PES and software safety lifecycle, a plan for the verification shall be establish concurrently with the development for the phase – IEC 6150 7.18.2.1. This goes well with XPs incremental development process. Information on the verification activities shall be collected and documented as evidence that the phase being verified has been satisfactorily completed – IEC 61508 7.18.2.4.

According to XP, design documentation has two purposes: to support maintenance and enhancements of the system, and to serve as input to the next stage of the development process [55]. The founders of XP are of the opinion that less documentation is needed for maintenance than for development. For maintenance the design information need only be on an overview level – additional details appear in the code. For development, detailed design information is needed as input to later stages, so the additional details must appear in the high-level design documentation to support the later stages. This need for detailed design documents is increased and enforced by the long time to coding in a waterfall project such as a project following IEC 61508. The same applies to the use of module responsibility, meaning that someone else will do the coding. The first objective of documentation in IEC 61508 is that all phases of the overall, E/E/PES and software safety lifecycles can be effectively performed - IEC 61508-1 requirement 5.1.1 The second objective is to specify the necessary information to be documented so that the management of functional safety, verification and the functional safety assessment activities can be performed effectively - IEC 61508-1 requirement 5.1.2. This involves an increased amount of documentation beyond what XP think is necessary. We see the need for documenting the process, but we will try to make use of the practices in XP to gather this information.

XP encourages programmers to use tests for documentation purposes. “The unit tests show how to create the objects, how to exercise the objects, and what the objects will do. This documentation, like the acceptance tests belonging to the customer, has the advantage that it is executable. The tests don’t say what we think the code does: they show what the code actually does**[27]**“ [28]. The requirements that all tests must run 100% at all times, ensures that the documentation from unit tests is kept up-to-date. Unit test can also ease the understanding of the code. If a programmer needs to change an unfamiliar piece of code, he will try to understand the code by inspecting the test cases.

Ron Jefferies, on of the developers of XP, says [28]:
“If there is a business need for a document, the customer should request the document in the same way that she would request a feature: with a story card. The team will estimate the cost of the document, and the customer may schedule it in any iteration she wishes.”

This statement indicates that XP is receptive to more documentation than user stories, CRC-cards, code, unit and acceptance tests. This procedure could also be used if some safety-document is needed. The customer or safety specialist should request the document with a story card.

In order to conform to IEC 61508s requirement for design document, the significant designs within an iteration can be documented. After an iteration these documents can be reviewed. The solutions to issues found at the review can be written as stories to the next iteration. As suggested by Grenning [17], the design document should be so high-level that usual modifications and bug fixes do not affect it. Another possibility is to write the high-level document at the end of the project. The documentation can guide the reader to the right part of the code for details. Documentation tasks can be planned into any iteration. One advice is to document what has been built rather then what is anticipated.

We recommend structuring the documents produced during the lifecycle to ease the traceability, and to make it possible to search for relevant information. It shall be possible to identify the latest revision (version) of a document or set of information according to IEC 61508-1 requirement 5.2.10. In addition, the documents should follow the guideline of IEC 61508-1 requirement 5.2.6. 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

We are of the opinion that the documentation we have suggested easily can follow these guidelines.

5.16 Summary

We have in Chapter 4 and this chapter argued that the requirements in IEC 61508 have been satisfied. Clauses 4, 5, 6 and 8 of Part 1, 2 and 3 in IEC 61508 state requirements for claiming conformance to the standard, documentation, management of functional safety, and assessment, respectively.

In part 1, clause 4 of IEC 61508 says: “to conform to this standard it shall be demonstrated that the requirements have been satisfied to the required criteria specified (for example safety integrity level) and therefore, for each clause or subclause, all the objectives have been met.” It is acceptable to order the software project differently from the organization of this standard, provided all the objectives and requirements of clause 7 in IEC 61508-3 are met according to IEC 61508-3 requirement 7.1.2.5. The different organization can for example be to use another software safety model.

We have proposed a new safety lifecycle for the development of the software. If the proposal is used, it should be specified during safety planning in accordance with clause 6 of IEC 61508-1 and IEC 61508-3 requirement 7.2.1.

To comply with IEC 61508, each phase of the overall safety lifecycle model describes technical requirements that must be carried out before implementation of these technical requirements is realizable.

1. Existing safety-related activities and their resulting documentation need to be reviewed in order to determine if all lifecycle phases are properly addressed.
2. The objectives for each phase have to be described.
3. Persons responsible for achieving the objectives need to be appointed. They must be competent, well trained and aware of their responsibilities.
4. Data and documentation necessary to meet the requirements need to be identified and located.
5. Information flows need to be realized in order to ensure that the required data are available where and when necessary.

Table 14 summaries the adoption of the XP practices (described in Chapter 2) that we have discussed in Chapter 4 and in this chapter.

Table 14 Extreme Programming adoption

Ref to Chap. 2	Extreme Programming	Adoption status	Comments
P1	The planning game is used to create project plans	Fully adopted	For each iteration the planning game is carried out, and the project plan can constantly be refined.
P2	The project team is traveling light	Not adopted	Not possible to travel light when developing safety-related software, but the proposal presented in this report leads to a “lighter” development strategy.
P3	User stories are written	Fully adopted	Complementary comments are written to each user story. Formal requirements are written if required.
P4	Release planning creates the schedule	Fully adopted	One of the purposes with the planning game is to define the set of features required for the next release.
P5	Make frequent small releases	Not adopted	Since the integration of hardware and software is a comprehensive process it is likely to make only a small number of releases.
P6	The Project Velocity is measured as a metric	Fully adopted	This shows how fast the work is getting done. It is also possible to learn from the estimation.
P7	Iteration planning starts each iteration.	Fully adopted	For each iteration the planning game is carried out. Iteration planning is a part of the planning game.

P8	Move people around	Partly adopted	Assumes the people have the right competence for the assigned task.
P9	A stand-up meeting starts each day	Fully adopted	This practice does not conflict with any of the requirement in IEC 61508.
P10	Fix XP when it breaks	Partly adopted	If the changes does not conflict with IEC 61508.
P11	Accepted responsibility	Fully adopted	The developer accept the responsibility, the management does not force them.
P12	Planning for priorities	Fully adopted	The most critical and important tasks are implemented first.
D1	Small initial investment	Not adopted	To have a stable system, the initial investment have to be considered thoroughly and have to be comprehensive so that later supplement are not too costly.
D2	Simplicity	Fully adopted	Simplicity is one of the most important aspects when developing safety-related system.
D3	Choose a system metaphor	Not adopted	The system metaphor is too vague for a safety-related system.
D4	Use CRC cards for design session	Fully adopted	We believe that CRC-card is suitable in safety-related system, but also acknowledge the possibility of using UML use cases.
D5	Create spike solutions to reduce risk	Partly adopted	The spike solutions have to be somewhat more formal.
D6	No functionality is added early	Not adopted	To be able to have a stable system, some functionality has to be added early.
D7	Refactor whenever and wherever possible	Partly adopted	
C1	On-site customer	Fully adopted	Customer and safety specialist on the development team will possibly be a significant contribution to the XP team.
C2	Code must be written to agreed standards	Fully adopted	The same requirement can be found in IEC 61508.
C3	Code the unit test first	Fully adopted	IEC 61508 lets the verification procedures be performed parallel with the development.
C4	All production code is pair programmed	Fully adopted	We believe pair-programming will improve the code.
C5	Only one pair integrates code at a	Fully adopted	The integration regards the software integration.

	time		
C6	Integrate often	Partly adopted	Software integration can often be performed, but not hardware and software integration.
C7	Use collective code ownership	Fully adopted	Collective code ownership is adopted assumed qualified people changes the code.
C8	Leave optimization till last	Partly adopted	The adoption depends which part of the code the optimization affect. Safety-critical code should only be changed for good reasons.
C9	No overtime	Fully adopted	Does not conflict with the requirements in IEC 61508.
T1	All code must have unit tests	Fully adopted	IEC 61508 requires specified tests for every module.
T2	All code must pass all unit tests before it can be released	Fully adopted	According to IEC 61508 every test have to be passed.
T3	When a bug is found, a new test must be created	Fully adopted	This allows discovering early if the bug reappears in the future.
T4	Write test before refactoring	Fully adopted	The new tests make the refactoring less risky.
T5	Acceptance test are run often and the score is published	Fully adopted	The score shows the status of the project.
T6	Other tests can be used	Fully adopted	IEC 61508 suggests other tests besides module and integration tests.

6 Conclusion

In this report, we have seen that Extreme Programming, as described by Beck [1], is not suitable for developing safety-related system if conformance to IEC 61508 shall be met. The most significant differences are: development cycle, design methods, documentation, scope, although the most important difference is the focus on safety. XP uses an evolutionary approach suited for vague and changing requirements, and the system is developed in small increments. IEC 61508, on the other hand, assumes that every requirement is defined in the beginning of the development. Therefore the V-model is suitable. This assumption also makes it possible to plan the design up-front while XP continually evolve the design. XP recognizes the cost and time of updating documentation, and prefer to use code and tests as documentation to avoid these drawbacks. They use face-to-face communication in place of written documentation wherever possible. IEC 61508 on the other hand, uses documentation to assessment in addition to communication purposes. Every phase of the lifecycle, even to decommission, shall be documented. XP covers only development and maintenance of the system. The vital difference is, as said earlier, that XP lack safety focus. For instance, hazard and risk analysis is vital to a safety-related system and have to be carried out. Despite these differences, XP includes a number of practices that may have a beneficial effect on the development of safety-related system.

The development of safety-related software and systems need additional levels of skill in addition to those needed by other types of systems. Safe systems require time, effort, and special knowledge and experience. It is of vital importance to have a methodological approach to the complete development and assessment process when producing safety-critical systems. This is not to say that it is impossible to produce safe systems with XP, but XP offers no way of knowing that an adequate level of safety has been achieved. Neither does it provide the basis for convincing anyone outside the development team, e.g. a certification authority, that safety has been achieved. One of the most important aspects of a methodological approach to achieving safety is to have a definition of the process. IEC 61508 uses the safety lifecycle as a framework to structure its requirements. It is a basic requirement of the standard that a similar lifecycle is used to structure the activities relating to the development of the system. Compliance to the standard exonerates users of any blame in the event of a safety problem. Assessors require a demonstration of safety, and proof of conformance to the standard may only be one contributor, though perhaps a significant one, to this. Therefore, we suggest following the safety lifecycle except for phase 9, the software safety lifecycle. What we have provided are conjectures related to how the practices of XP *may* contribute to the development of the software part of safety-related system. We proposed a modified software safety lifecycle. This model explicit specifies iterations, and makes it possible to adopt possible changes. The system is based on a stable architecture that should not be modified, but minor requirement changes are possible with the use of refactorings. It should be noted that every phase in the IEC software safety lifecycle is covered in the proposed model.

The need to adapt to rapidly changing requirements is impacting software engineering, as far as flexibility of software development is concerned. XP is a development

methodology that was specifically conceived to work in the face of vague and changing requirements. The proposal in this report opens for the possibility for some changing requirements. To be able to adapt the changing requirements we need to modify the code by refactorings. IEC 61508 have strict, time-consuming procedures for modification of the software. These modification procedures are, on the other hand, meant for modification after the software design and development are finished, in contrast to refactoring that is performed during the development. A unit test has to be written before any refactoring, and this test detects most of the failures that can arise due to refactoring. Therefore, we believe that refactoring is possible in safety-related systems. If hardware is available during the software development, it may be possible to deliver the system incrementally. Furthermore, the short revision times aid in project tracking and scheduling.

In addition to the proposed software lifecycle, we have some recommendations to other phases of the overall safety lifecycle as well. Hazard and risk analysis is carried out early in the overall safety lifecycle. We recommend having the project manager and some safety specialists that are a part of the XP team, present during this analysis. In that way the programmer have the possibility to ask persons at any time about important safety aspects.

One of the most effective rules for making things safer is simplicity. The simpler the system is the more likely is it that errors will not be introduced. In addition the system is easy to understand and structure. Simplicity is one of the values in XP, and the team should always ask themselves: “What is the simplest thing that could possible work?” This mutual view on the importance of simplicity is one of the reasons why XP can be attractive to safety-related system.

IEC 61508 and XP have different view on the amount of documentation that is needed. We see the importance of documentation when developing safety-related system. If an XP project gets into trouble, there are few fall back scenarios, as the project may end up with an undocumented system that is hard to modify and not prepare for the changes of tomorrow. Therefore we suggest writing all documentation that is necessary for communication and for carrying out a safety assessment. Any demand for specific documents can be captured as user stories. If a safety case should be carried out, it should be possible to identify the reasons why the system is believed to be safe. In many situations, the safety case will be the major deliverable to the certification process or to an independent assessment. In practice, the arguments will reflect design information, reliability calculations, safety analyses, and perhaps proofs of program properties. The goal is to achieve assurance, and it is therefore important to have control over the process and have documentation of the process. Even though we recommend writing more documentation, we make use of some of XPs techniques to make the process easier. For example, user stories are collected to form the requirement document.

Safety is a system issue and a human factors issue rather than merely a software issue. The safety of software can not be evaluated by looking at the code alone. Safety can only be evaluated in the context of which the system operates. Total safety cannot be

guaranteed, so it is important to calculate and understand the risk involved in a given circumstance and define a 'tolerable risk'. One have to identify the risk, determine how and by how much to reduce the risks, justify the decisions, reduce the risks, demonstrate this reduction, and accept responsibility for all decisions. Safety is about reducing the risk. Therefore, safety does not depend on the quality of the software alone, but on a deeper study of potential hazards and their likelihood. A risk-based approach means not merely following a procedure and assuming that "safety" will result. Compliance to a standard gives an approval that the software is developed in a certain way and that it can be trusted up to a certain level of safety. In IEC 61508, techniques and measures shall be selected according to the safety integrity level. But following these techniques do not ensure that the software is safe enough for that integrity level [6].

Even though the safety culture is conservative concerning changes and new methods, IEC 61508 has been conceived with a rapidly developing technology in mind. The framework is sufficiently robust and comprehensive to cater to future developments. This is one of the reason that some of the practices of XP could be adopted. Our proposal aims at safety integrity level 1 and 2. These levels do not prevent the use of the practices included in our proposal. In addition, they do not require use of formal methods in the development.

Most of the XP guidelines were adopted in our proposal. Ten out of twelve key practices in XP where either fully or partly adopted. The introduced practices are: the planning game, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-hour week, on-site customer, and coding standard. The practices that are not adopted are small releases and metaphor. One solution for adopting XP is to suggest small changes, introduce simple disciplines and let their benefits speak for themselves.

Our approach to develop safety-related system has many advantages over the practices described in IEC 61508. The customer is involved in the development of the software, and makes it possible to have early feedbacks. Poorly understood requirements can be early discovered, and there will be many opportunities to make corrections due to the rapid feedback. Since the customer is part of the XP team, the developer can always ask the customer if something is vague or unclear. XP's iterative development can help the team determine how fast it could go, and give management the feedback they needs. It is easier to control the development by making many minor adjustments. This does not only apply to the manger, but the developer can steer the development based on the learning from every increments. In addition, the iterations add flexibility to the system since requirements can change, and the most important requirements are implemented first. Planning sessions can help to spread the design throughout the team. We believe that the proposal stated in this report can lead to improved quality, faster development, better predictability, greater job satisfaction, and achieve an appropriate level of safety with as low cost as possible.

It can be argued that the proposal not complies with XP, since only some of the practices are adopted. We acknowledge that the proposal does not follow XP in every part. But one of the major practices in XP, stated by Kent Beck, is as follows [2]: "By being part of an

Extreme team, you sign up to follow the rules. But they're just rules. The can change the rules at any time as long as they agree on how they will access the effects of the change." This statement opens for changes in XP, and our proposal can be categorized as safety-XP. Therefore we feel free to conclude that safety-XP can be used and will possibly bring gain to the development process.

7 Further work

The literature search revealed the need for more research on consolidating the recommendations stated in this report. It would be particularly interesting to see the result of a case study or experiment based on the stated proposal.

Some topics of interest where briefly or not mentioned in the report, and can be subject to further review. The next paragraphs points out some of these areas.

XP is a methodology for small-sized teams. Thus we have focused on small development teams in this report. Safety-related systems though, may require bigger teams. Therefore it could be useful to investigate the possibility to scale up the team. Jacobi and Rumpe **[26]** identified several obstacles against scaling up XP. The most important are lack of documentation, lack of stable interfaces, and lack of stable requirements. If following the proposals in this report, all these aspects are more comprehensive and stable. Consequently, scaling up XP will probably be easier in this case than it is in a normal XP project.

We have discussed the benefits of unit testing, and have suggested use of testing framework, e.g Junit and CppUnit. No certification of these framework have been carried out. It would thus be interesting to evaluate it. Another interesting issue that should be investigated further is the possibility of developing a unit-testing framework suited for testing of safety-related system.

One of the reasons why XP wants to refactor is redundant software. In safety-related system however, redundancy is widely used. All forms of fault tolerance are achieved by some form of redundancy. One research area can be to explore if hardware redundancy can simplify the software development in such a way that the practices of XP is easier to perform.

Although this report has focused on the technological aspects of the problem, there are also managerial and organizational issues that should be considered. Concentrating only on technical issues and ignoring managerial and organizational deficiencies will not result in safe systems. Examples of managerial and organizational that can be of interest are; diffusion of responsibility and authority can leave the burden of ensuring safety on individuals who do not have the authority to carry out their responsibilities, and safety personnel with a lack of independence. When safety personnel report to the same authority that is responsible for budget and schedule considerations, which can be the case in an XP project, there is some likelihood that schedule or budgetary pressures will override safety.

Appendix A: Definitions

Equipment under control: “Equipment, machinery, apparatus or plant used for manufacturing, process, transportation, medical or other activities” [25].

Failure: “Termination of the ability of a functional unit to perform a required function” [25].

Functional safety: “Part of the overall safety relating to the EUC and the EUC control system which depends on the correct functioning of the E/E/PE safety-related systems, other technology safety-related system and external risk reduction facilities” [25].

Harm: “Physical injury or damage to the health of people either directly or indirectly as a result of damage to property or to the environment” [25].

Hazard: “The capability to do harm to people, property or the environment, is termed a hazard” [54].

Impact analysis: “Activity of determining the effect that a change to a function or component in a system will have to other functions or components in that system as well as other systems” [25].

Integration testing: “Is a set of procedures designed to verify whether a given assembly of components (be they systems, software or tools) does indeed satisfy the requirements of an organization” [53].

Low complexity E/E/PE safety-related system: “E/E/PE safety related system in which [25]:

- The failure modes of each individual component are well defined
- The behavior of the system under fault conditions can be completely determined.”

Mode of operation: “Way in which a safety-related system is intended to be used, with respect to the frequency of demands made upon it, which may be either [25]:

- **Low demand mode:** where the frequency of demands for operation made on a safety-related system is no greater than one per year and no greater than twice the proof test frequency
- **High demand mode or continuous mode:** where frequency of demands for operation made on a safety-related system is greater than one per year or greater than twice the proof check frequency”

Pair programming: “All production code is written with two people looking at one machine, with one keyboard and one mouse” [1].

Programmable electronic (PE): “Based on computer technology which may be comprised of hardware, software, and of input and/or output units” [25].

Refactoring: “Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility” [1].

Reliability: The probability of a component, or system, operating as defined within its specification over a given period of time under a given set or operating conditions [54].

Risk: “Combination of the probability of occurrence of harm and the severity of the harm” [25].

Risk analysis: “Identification of situations that could endanger human life or the environment” [54].

Safe system: A system is safe if it can’t (it acceptally unlikely to) cause absolute harm, or fail to prevent it when it is intended to do so [42].
Designated system that both [25]:

- Implements the required safety functions necessary to achieve or maintain a safe state for the EUC; and
- Is intended to achieve, on its own or with other E/E/PE safety-related systems, other technology safety-related systems or external risk reduction facilities, the necessary safety integrity for the required safety functions.

Safety analysis: “The process of assessing the safety of a system by looking at the associated hazards and the methods used by the system to cope with them” [54].

Safety function: “Function to be implemented by an E/E/PE safety-related system, other technology safety-related system or external risk reduction facilities, which is intended to achieve or maintain a safe state for the EUC, in respect of a specific hazardous event” [25].

Safety integrity: “Freedom from flaws or corruption which could compromise safety” [42]. “Probability of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time” [25].

Safety integrity level: “Discrete level (one out of four) for specifying the safety integrity requirements of the safety functions to be allocated to the E/E/PE safety-related systems, where safety integrity level 4 has the highest level of safety integrity and safety integrity level 1 has the lowest” [25].

Safety-related system: designated system that both [25]:

- “Implements the required safety functions necessary to achieve or maintain a safe state for the EUC
- Is intended to achieve, on its own or with other E/E/PE safety-related systems, other technology safety-related systems or external risk reduction facilities, the necessary safety integrity for the required safety functions.”

Software integration: ”Is the practice of assembling a set of software components/subsystems to produce a single, unified software system that supports some need of an organization” [53].

Testing: “The process used to verify or validate a system or its components” [54].

Validation: “The process of determining that a system is appropriate for its purpose” [54].
“Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled” [25].

Verification: “The process of determining that a system, or module, meets its specification” [54]. “Confirmation by examination and provision of objective evidence that the requirements have been fulfilled” [25].

Appendix B: Abbreviation

BDUP	big design up front
EUC	equipment under control
E/E/PE	electrical/electronic/programmable electronic
E/E/PES	electrical/electronic/programmable electronic system
FMEDA	failure modes, effects and diagnostic analysis
IEC	International Electro-technical Commission
SIL	safety integrity level
SIS	safety instrumented system
SRS	safety-related system
UML	Unified Modeling Language
XP	Extreme Programming

References

[1] K. Beck, E*xtreme Programming Explained: Embrace Change*, Boston: Addison Wesley, 2000.

[2] K. Beck, Embracing Change with Extreme Programming. *Computer*, Vol. 32 No. 10, October 1999.

[3] K. Beck, E. Gamma, 1999. *JUnit* [online]. Place of publication: Available from: Junit.org. Available from: http://www.junit.org [Accessed date: 05.03.2002].

[4] R. Bell, IEC 61508: Functional safety of electrical/Electronic/programme Electronic safety-related systems: Overview. *Control of Major Accidents and hazards Directive (COMAH) – Implications for Electical and Control Engineers*, Ref. No 1999/173, IEE Colloquium, 1999.

[5] R. Bell, T. Stålhane, T. Malm, J. Jacobsen. Functional safety of programmable electronic systems – evaluation according to IEC 61508, *Seminar, Stockholm.* February 1999.

[6] P. Bennett, The March Towards Standards in Safety Related Systems. *Computers and Safety. A First International Conference on the Use of Programmable Electronic Systems in Safety Related Application*, 1989.

[7] S. Brown, Overview of IEC 61508: Design of electrical/electronic/programmable electronic safety-related systems. *Computer & Control Engineering Journal*, February 2000.

[8] N. Churcher, C. Cerecke, GROUPCRC: exploring CSCW support for software engineering, *Proceedings., Sixth Australian Conference on*, 1996.

[9] A. Cockburn, L. Williams. The cost and benefits of pair programming. *Extreme Programming Examined.* Addison-Wesley, 2001.

[10] C. Collins, R. Miller, *XP Distilled* [online]. Place of publication: RoleModel Software. Available from: http://www.rolemodelsoft.com/articles/xpCorner/xpDistilled.htm [Accessed date: 21.01.2002].

[11] A. Deursen, Program Comprehension Risks and Opportunities in Extreme Programming, *Reverse Engineering, 2001. Proceedings. Eight Working Conference on*, 2-5 October 2001.

[12] M. Fowler, *Refactoring.* USA: Addison-Wesley Longman, Inc, 1999.

[13] M. Fowler, *UML Distilled second edition.* USA: Addison-Wesley, 2000.

[14] R. Glass, Extreme Programming: The good, the bad, and the bottom line. *IEEE Software*, November/December 2001.

[15] D. Gluch, C. Weinstock, 1997. *Workshop on the State of the Practice in Dependably Upgrading Critical Systems* [online]. Place of publication: SEI. Available from: http://www.sei.cum.edu/pub/documents/97.reports/pdf/97sr014.pdf [Accessed date: 08.05.2002].

[16] W. Goble, J. Bukowski, Extending IEC61508 Reliability Evaluation Techniques to Include Common Circuit Designs Used in International Systems. *Proceedings Annual Reliability and Maintainability Symposium*, 2001.

[17] J. Grenning, Launching Extreme Programming at a Process-Intensive Company. *IEEE Software*, November/December 2001.

[18] K. Heel, B. Knegtering, A. Brombacher, Safety lifecycle management. A flowchart presentation of the IEC 61508 overall safety lifecycle model. *Quality and Reliability Engineering International*, Vol. 15 No. 6, 1999.

[19] HISE, 2002. *Safety critical mailing list* [online]. Place of publication: The High Integrity Systems Engineering Group (HISE), University of York. Available from: http://www.cs.york.ac.uk/hise/hise4/frames9.html [Accessed date: 12.04.2002].
[20] IEC, 2002. *International Electrotechnical Commission.* Available from: www.iec.ch [Accessed date: 24.03.2002].
[21] IEC, 2001. *IEC 61508 Sector A: Scope* [online]. Place of publication: IEC. Available from: http://www.iec.ch/61508/Scope.htm [Accessed date: 24.03.2002].
[22] IEC, 2001. *IEC 61508 Sector B: Position in overall standards framework* [online]. Place of publication: IEC. Available fromhttp://www.iec.ch/61508/Position.htm [Accessed date: 24.03.2002].
[23] IEC, 2001. *IEC 61508 Sector C: Legal status and standard compliance* [online]. Place of publication: IEC. Available from: http://www.iec.ch/61508/Compliance.htm [Accessed date: 24.03.2002].
[24] IEC, 2001. *IEC 61508 Sector D: Key Concepts* [online]. Place of publication: IEV. Available from: http://www.iec.ch/61508/Concepts.htm [Accessed date: 24.03.2002].
[25] International Electrotechnical Commission, *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related system*. Geneva, 1998.
[26] C. Jacobi, B. Rumpe, Hierarchical XP: Improving XP for large-scale projects in analogy to reorganization processes. *Extreme Programming Examined.* Addison-Wesley, 2001.
[27] R. Jeffries, Extreme Testing. *Software Testing & Quality Engineering*, March/April 1999.
[28] R. Jeffries, 2001. *Essential XP: Documentation* [online]. Place of publication: XProgramming.com. Available from: www.xprogramming.com/xpmag/expDocumentationinXP.htm [Accessed date: 21.01.2002].
[29] R. Jeffries, 2001. *XP and Reliability* [online]. Place of publication: XProgramming.com. Available from: www.xprogramming.com/xpmag/Reliability.htm [Accessed date: 21.01.2002].
[30] R. Jeffries, A. Anderson, C. Hendrickson. *Extreme Programming Installed.* USA: Addison-Wesley, 2000.
[31] P. Johannessen, C. Grante, A. Alminger, U. Eklund, J. Torin. Hazard analysis in object oriented design of dependable systems. *Dependable Systems and Networks.* Proceedings. The International Conference on, 2001.
[32] J. Kivi, D. Haydon, J. Hayes, R. Schneider, G. Succi, Extreme Programming: a university team design experience. *Electrical and Computer Engineering*, 2000 Canadian Conference on, Vol. 2, 2000.
[33] H. Krosigk, Functional safety in the field of industrial automation: The influemce of IEC 61508 on the improvement of safety-related control systems. *Computing & Control Engineering Journal*, February 2000.
[34] N. Leveson, 2001. *The difference between software safety and hardware safety* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/differ.shtml [Accessed date: 12.04.2002].
[35] N. Leveson, *Safeware: System safety and computers.* Addison-Wesley Publishing Company, Reading MA, 1995.
[36] R.Martin, IEEE Software, *extreme Programming Development through Dialog*, Vol. 17 No. 4, July/August 2000.

[37] J. Nawrocki, B. Walter, A. Wojciechowski, *Euromicro Conference Proceedings. 27th*, Toward Maturity Model for extreme Programming, 4-6 September 2001.
[38] O. Nordland, Understanding safety integrity levels. *The safety-critical Systems Club Newsletter*, Vol. 11, No. 1, September 2001.
[39] J. Nosek, *The Case for Collaborative Programming*, Communications of the ACM, 1998.
[40] C. Poole, J. Huisman, Extreme Maintenance. *Proceedings. IEEE International Conference on Software Maintenance (ICSM'01)*, 2001.
[41] F. Redmill, IEC 61508: Principles and use in the management of safety. *Computer & Control Engineering Journal*, February 2000.
[42] F. Redmill, T.Anderson, *Safety-critical systems*. London:Chapman & Hall, 1993.
[43] L. Rierson, Changing Safety-Critical Software. *IEEE AESS Systems Magazine*, June 2001.
[44] Safeware Engineering Corporation, 2001. *Hazard Analysis* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/hazard-analysis.shtml [Accessed date: 12.04].
[45] Safeware Engineering Corporation, 2001. *Preliminary Hazard Analysis* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/prelim-analysis.shtml [Accessed date: 12.04].
[46] Safeware Engineering Corporation, 2001. *Software Hazard Analysis* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/subsys-analysis.shtml [Accessed date: 12.04].
[47] Safeware Engineering Corporation, 2001. *Design for Safety* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/design.shtml [Accessed date: 12.04.2002].
[48] Safeware Engineering Corporation, 2001. *Organization and Management* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/orgmanage.shtml [Accessed date: 12.04.2002].
[49] Safeware Engineering Corporation, 2001. *Accidents* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/accidents.shtml [Accessed date: 12.04].
[50] Safeware Engineering Corporation, 2001. *Overview of a Software Safety Approach* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/approach.shtml [Accessed date: 12.04].
[51] Safeware Engineering Corporation, 2001. *Conclusions* [online]. Place of publication: Safeware Engineering Corporation. Available from: http://www.safeware-eng.com/software-safety/conclusions.shtml [Accessed date: 12.04].
[52] D. Smith, 1996. *Frequently asked questions* [online]. Place of publication: D. Smith. Available from: http://www.dnsmith.com/SmallFAQ/PDFfiles/index.html [Accessed date: 12.04.2002].
[53] V. Stavridou, Integration standards for critical software intensive systems. *Institute of Electrical and Electronic Engineers, Inc*, 1997.

[54] N. Storey, *Safety-critical computer systems*. USA: Addison-Wesley, 1996.
[55] G. Succi, M. Marchesi, *Extreme Programming Examined*. USA: Addison-Wesley, 2001
[56] A.Summers, Draft IEC 61508 Target Safety Integrity Levels, *Institute of Instrumentation and Control Australia Inc*, Vol. 14, No. 3, July 1999.
[57] J. Thompson, M. Heimdahl, An Integrated Development Environment for Prototyping Safety Critical Systems. *Rapid System Prototyping, 1999. IEEE International Workshop on*, 1999.
[58] K. Van Heel, B. Knegtering, A. Brombacher, Safety lifecycle management. A flowchart presentation of the IEC 61508 overall safety lifecycle model. *The Netherlands Eindhoven University of Technology*, 1999.
[59] D. Wells, 1999. *The Rules and Practices of Extreme Programming* [online]. Place of publication: extremeprogramming.org. Available from: http://www.extremeprogramming.org/rules.html [Accessed date: 01.02.2002].
[60] L. Williams, R. Kessler, W. Cunningham, R. Jeffries. *Strenghten the Case for Pair Programming*, Place of publication: IEEE Software. Available from: http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.pdf. [Accessed date: 13.03.2002].
[61] L. Williams, R. Kessler. *The collaborative Software Process.* International Conference on Software Engineering 2000. Limeric, Ireland. Available from: http://www.cs.utah.edu/~lwilliam/Papers/ICSE.pfd [Accessed date: 05.03.2002].
[62] L. Williams, R. Kessler, W. Cunningham, R. Jeffries, Strengthening the Case for Pair Programming, *IEEE Software*, Vol. 17 No. 4, July/August 2000.