From a Single Discipline Risk Management Approach to an Interdisciplinary One: Adaptation of FMEA to Software Needs

Susanne Hartkopf
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6, 67661 Kaiserslautern, Germany
Phone +49 6301 707 238; Email hartkopf@iese.fraunhofer.de

Abstract

Risk management has been identified as a vitally important project management task. Hence, many risk management approaches have been developed. Unfortunately, most of them deal with the risks of a single discipline only, meaning disciplines in which either software or non-software products are developed. In contrast, nowadays many projects are highly interdisciplinary undertakings in the sense that newly developed conventional non-software products are enhanced by software. With the advent of software, many additional risks have emerged. In this paper, the differences between software and non-software products are investigated. From these differences, consequences for interdisciplinary projects are derived. It is indicated how an interdisciplinary risk management approach can cope with the consequences. An answer is given to the question of how to achieve such an interdisciplinary approach. One possible solution is presented here as an adaptation of the Failure Modes and Effects Analysis, a single discipline approach, to the needs of software. This paper is an extension of a position paper presented at the STEP2003 Workshop of Interdisciplinary Software Engineering.

1. Introduction

This paper is an extension of a position paper presented at the second Workshop on Interdisciplinary Software Engineering (WISE) held during the tenth International Workshop Software Technology and Engineering Practice (STEP) in Amsterdam, The Netherlands, in September 2003. The position paper was called “Interdisciplinary Risk Management”, and the position statement was: *In order to meet the challenge of interdisciplinary projects in the German automotive industry, it is necessary to adapt the established* risk *management technique FMEA, which originates from the mechanical and electrical engineering field, to a situation, where software and hence software engineering plays a more and more prominent role in automobiles and their production processes* [1]. It contributed to the topic of WISE 2002 in that it draws on a technique that is used in a discipline cognate to software engineering in order to transform software engineering into a more holistic, interdisciplinary activity that breaks down rigid barriers between disciplines and processes [2], [3].

In [3], a framework is presented, that identified, first, seven groups of issues that are related to contemporary software engineering practice and research (e.g., information explosion, procurement decision, risk identification and management) and, second, thirteen disciplines cognate to software engineering (e.g., civil engineering, systems engineering, production engineering). For each group of issues, techniques or models from these cognate disciplines are identified that are recognized as transferable to software engineering along with the level of diffusion. This level describes to which extent a transformation from the cognate discipline to software engineering has already taken place.

In order to classify the disciplines (mechanical and electrical engineering) and technique (risk management technique FMEA) from the position statement into the table of the above mentioned framework, some explanation is needed, because the classification is not straightforward. Certainly, risk management was identified as one group of issues of current software engineering practice and research, but mechanical or electrical engineering were not explicitly regarded as a cognate disciplines. Considering the automotive context of the position statement, the most cognate discipline seems to be the discipline of production engineering. This assignment can be supported, since FMEA as described in [4], [5] defines a risk management process that covers the same life cycle phases as described for production engineering in [3]. The level of diffusion of FMEA can be stated between 3 and 4. That means from “best practice in some organizations, perhaps covered by frameworks CMM, SWEBOK, RUP …" to "waiting on standards being developed and accepted". An example for best practice in some organization is described in [6]. Here, the application of FMEA for software-intensive components at Siemens is exemplified. However, quite a lot of organizations encountered problems when applying FMEA in projects in which they produce products that were formerly pure non-software products and are now enhanced by software. These organizations look for unified guidelines on how to perform FMEA in interdisciplinary projects.

After the explanation of the context of the paper, in the following, the structure of the paper as well as its extension in comparison to the position paper is explained.

Chapter 2 describes the gap that arises in interdisciplinary projects due to the fact that project management has no means to cope with new demands in such projects and risk management approaches are traditionally designed for single discipline projects. Chapter 3 and 4 elaborate thoroughly on the question of the position paper, *Why do we need an interdisciplinary risk management approach?* In Chapter 3, the differences between software and non-software products are worked out. Chapter 4 describes the consequences with respect to products combining software and formerly non-software products, the processes in the life cycle of such products, and the people involved in such interdisciplinary projects. The question from the position paper, *Why is risk management a useful means to bridge the gap between disciplines?* is investigated in depth in Chapter 5. Chapter 6 explains how an interdisciplinary risk management can be achieved, whereas in Chapter 7 one possible solution for an interdisciplinary risk management approach is described. Chapter 8 previews the expected benefits for the different disciplines involved. Chapter 9 closes the paper with a summary.

2. The gap in interdisciplinary projects

In the context of this paper, interdisciplinary projects are projects in which engineering products are developed that were formerly pure non-software products and are now enhanced by software. That means, in the development of these combined products, components with different features, processes from more than one engineering discipline, and project staff with different educational backgrounds are involved and must be integrated.

In interdisciplinary projects, a gap between project management and its vitally important activity of risk management can be observed. On one side of the gap, risk management approaches are traditionally developed for a single discipline, and on the other side, project management has few means to cope with the challenges in interdisciplinary projects. In order to close the gap, an extension of single discipline risk management approaches to the needs of interdisciplinary projects is a potential solution.

This extension is necessary as, on the one hand, market trends show a general shift from single discipline to interdisciplinary projects, and, on the other hand, the media report on the quality problems of combined products almost every day. For example, in the automotive industry, more and more functionality is and will be provided by software, in addition to conventional mechanical and electrical components. Independently of any discipline, risk management is recognized as a very important project management activity to successfully perform a project. So far, project managers apply – if at all – existing, single discipline risk management techniques to their projects. They use approaches they are familiar with, neglecting the fact that their projects have mutated into interdisciplinary projects. They can hardly do better, since risk management techniques that cover the needs of interdisciplinary projects are still missing.

In the following, the essential differences between software and non-software products are examined. This is done to better understand the difficulties encountered in interdisciplinary projects.

3. Essential differences between software and non-software

In the following, the differences between software and non-software are thoroughly investigated. Since "non-software" is an awkward term to use, the term hardware is introduced, it is used for non-software products such as items made from steel or metal.

In order to figure out the differences between software and hardware, a literature review was performed. The results presented here are mainly taken from Beizer [7] and the well-known article of Brooks [8]. Nevertheless, if other papers contributed valuable aspects, then they are referenced as well.

The main differences between software and hardware can be reduced to invisibility, intangibility, non-applicability of natural constraints, changeability, conformity, and complexity. Thereby, Brooks' four essences of software are extended by intangibility and the non-applicability of natural constraints. This is done because these two create many consequences when comparing software engineering with the other engineering areas.

The six main differences form a quasi-deduction chain. In Figure 1, a quasi-deduction chain of the differences between software and hardware is depicted. Quasi-deduction here means that the line of the deduction might not be as strict as the arrows indicate. Nevertheless, from these six differences, the major consequences for interdisciplinary projects can be derived. In the following, each of the differences is defined, their characteristics for software and hardware are worked out and, if seen as necessary, explained with examples.

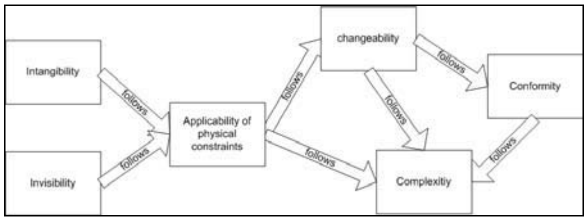

Figure 1: Quasi-deduction chain of the differences between software and hardware

3.1 Invisibility and intangibility

Invisibility and intangibility are somehow similar concepts. Nevertheless, both are considered, because in the remainder of the paper, both concepts are referenced depending on what is easier to understand.

Definition "Invisibility": Not visible, not able to be perceived by the eye [9].

Definition "Intangibility": Incapable of being perceived by touch; impalpable [9], [10].

Hardware is visible and can be touched. Hardware has a material embodiment, for example, a punch or a hammer. The action or movement of hardware is normally directly observable without any special means.

Software is invisible [8] and intangible [11]. To be more precise, the intangibility of software is manifested in two ways. First, software entities do not have a material embodiment [11], [12], except being presented to human beings by numbers, letters, or graphics on paper or on the monitor screen. Second, the visibility of software code when in action is normally not observable or solely indirectly [13] and just by means of a computer [12].

Straight consequence: From these two differences, the applicability or non-applicability, respectively, of natural constraints to software and hardware follows [12].

3.2 Non-applicability of natural constraints

Definition: Natural constraints are the natural laws and the properties of physical material.

Hardware is subject to natural constraints. Thus, these enforce discipline on the design, construction, and modification of hardware [14].

Software. With the advent of software, a new way of thinking was introduced. The intuitive understanding of the physical world as human beings had perceived it up to that point has been turned up-side down. Software code is stored in electronic bits and no natural law or property of physical material governs the software engineer when designing, constructing, or modifying the software [14]. That means, on the other hand, that discipline is brought into the design, construction, and modification not through natural constraints, but through the capability of the individual software engineer or a group of engineers [11]. If they are disciplined and use proper tools, methods, processes, and architecture, there is a chance that software is engineered in a sensible way.

Straight Consequences. From the applicability or non-applicability of natural constraints, two consequences can be derived. First, it makes a difference in the changeability of software and hardware (see Section 3.3). Second, it has an impact on the degree of complexity which can be reached in software and hardware [14], [12] (see Section 3.5).

3.3 Changeability

Definition: Changeability is understood here as a superior term to four aspects associated with software's and hardware's ability to change throughout their entire life cycle. (1) *Ease of modification:* The ease with which hardware or software can be modified before mass production. (2) *Ease of extendability*[1]: The ease with which software and hardware from an entire release and series, respectively, can be modified to increase functional capacity. (3) *Ease of flexibility*[1]: The ease with which software and hardware from an entire release and series, respectively, can be modified for use in applications or environments other than those for which it was specifically designed. (4) *Ease of maintainability*[1]*:* The ease with which software and hardware from an entire release and series, respectively, can be modified to correct faults, or improve performance or other attributes.

Hardware. For hardware, *(1) Ease of modification* is relatively easy. In the development phase, i.e., before the prototype is built and the build-up of the

[1] Definitions adhere to the definitions in [15].

infrastructure has started, the design of the new product is performed mostly in the heads of the engineers and on paper. In the further course of the life cycle, the following is true: the more of the prototype and the infrastructure for the manufacturing process is finished, the harder it gets to implement major changes. This explanation is true for the production of a building as well as of a car model.

(2) Ease of extendability is hard for hardware. Just think of the following example: a manufacturer of a rear-wheel drive car wants to extend it to an all-wheel drive. Most likely, this extension can not be done for cars already in the field. The infrastructure for production needs a redesign, or even worse, a new site for the plant. The line between ease of extendability and *(3) Ease of flexibility* is hard to draw for hardware. An example for flexibility could be the modification of a car toward an amphibian car. Nevertheless, ease of flexibility and extendability behave in similar ways.

(4) Ease of maintainability in the sense of correcting faults is hard for hardware. A good example for this are many recalls that are initiated in order to correct systematic faults for all cars in a series that are already sold. In contrast, recalls are rare in which the manufacturer wants to improve performance or other attributes. Such a maintainability action might be performed for cars in a series not yet built, but not for cars already on the street. Ease of maintainability in the sense of improving attributes such as performance does not really exist for cars and, therefore, maybe neither for the majority of hardware products.

Software. The perception by the general public is that changeability of software is high. The four aspects of changeability seem to be realized relatively easy. Major changes seem to be performed quickly and at seemingly low cost [14]. However, the consequences of the changes are often not considered carefully.

For *(1) Ease of modification* it is true, that a modification has always an impact on the software product, because the software developed before mass production is always the final product (except if prototyping was selected as process model). That means the basis for the future quality is already laid in the development phase. *(2) Ease of extendability and (3) Ease of flexibility* are seen by the general public as beneficial advantages of software. Why *(4) Ease of maintainability* is perceived as easy to perform is shown by the following example: A software user can easily maintain his software by installing a new service pack either down-loaded from a storage device or from the Internet. Then faults are corrected, and even performance might be improved.

The perception of "easy to perform" must be further explained. As long as the changes are performed only on the code, then a change seems to be easy to perform. However, often side effects to other parts of the code are not considered carefully, which may cause different, maybe severe failures. Additionally, in order to ensure traceability between the different software entities (i.e., requirements document or design document), the change should be performed universally in every entity. If a check of the potential side effects is conducted beforehand and the changes are performed universally, then the ease is immediately reduced.

Straight Consequences. Changeability has an impact on conformity (see Section 3.4) and complexity (see Section 3.5).

3.4 Conformity

Definition: "Compliance in actions, behavior, etc., with certain accepted standards and norms." [9]

Hardware: Due to the natural constraints, the degree of conformity of hardware is limited.

Software: Software is conformable for two reasons. First, it is perceived as the most conformable. Second, it is the most recent arrival on the scene [8]. Inconsistencies discovered in a late phase of hardware development are often adjusted by software, because it is perceived as easy to conform. The same is true for hardware that has been around for a long time. If this hardware is supposed to be enhanced, often software is selected for the enhancement and is made conform.

Straight Consequences: Conformity has an impact on complexity. If a product is highly conformable, then it is likely that complexity increases with every step of conformity.

3.5 Complexity

Definition: In [15] complexity is defined as the degree to which a system or component has a design or implementation that is difficult to understand and verify. In [10], complexity is defined as the state or quality of being intricate or complex, whereby intricate can be described as difficult to understand and complex can be described as made up of various [involved] interconnected parts. Both definitions emphasize that the difficulty in understanding indicates the degree of complexity. Therefore, the following measures for complexity, some found in literature, will be introduced, in order to facilitate the comparison between software and hardware: (1) *Proportionality of complexity*: Describes what effect the addition of new features on the system size has. [7], [8]. (2) *Intertwining*: Describes the inner structure of a system. Two forms of description are selected: The degree of decomposition and the clearness of graphical presentation. (2a) A*pplicability of the decomposition principle:* Describes to which extent a system can be dismantled. [7], [14] (2b) *The clearness of graphical presentation:* Describes how easily a graphical presentation can be understood. (3) *Functionality/complexity relation:* This relation describes the visible impact of functionality on complexity. [7]. (4) *Repeatability of similar elements:* This describes to which extent similar elements exist, physically or as software code, and how they are reused, in terms of software terminology invoked. [8] .

Hardware. In order to explain the nature of hardware, an example is used. The example relates to the design of an apartment building. *(1) Proportionality of complexity* can be best explained with the relation to costs: If the future builder-owner wants to have a building with one more floor, then costs increase in nearly linear proportion. That means in reverse, with respect to complexity, that complexity will not increase more than in linear proportion. Otherwise the architect would increase costs as well. The *(2a) Applicability of the decomposition* principle can be called high, because a building can be dismantled to its smallest not decomposable elements (e.g., rooms, floors, doors, ceilings, walls, windows, and so on). *(2b) Clearness of graphical presentation* of a building is easy to understand by looking at the blueprint. In general, hardware can be documented by literal engineering drawing. Concerning the *(3) Functionality/complexity relation,* it can be stated that more functionality for the user increases complexity for the user. For example, if a tenant wants a heating system and shutters with programmable heating times and opening/closing times respectively, then this increases the usage complexity for the tenant. *(4) Repeatability of similar elements* is high in a building. It consists of many elements that have the same functionality (e.g., doors), but are repeated in their material embodiment.

Software. *(1) Proportionality of complexity* of software increases more than in linear proportion. In most cases, a newly added element interacts with the other elements in some non-linear fashion and this increases the complexity of the whole much more than in linear proportion [8]. *(2a) Applicability of the decomposition principle* is also an objective in software engineering. However, other software engineering objectives, such as global data storage or the principle of abstraction [13] interfere with this objective [7]. *(2b) Clearness of graphical presentation* is mostly not given, due to several, generally directed graphs superimposed one upon another [8]. These graphs may represent flow of control, flow of data, time sequences, and even more (compare with all the different UML Diagrams). Additionally, these graphs are usually not planar and much less hierarchical [8]. The *(3) Functionality/complexity relation* of software is often an inverse relation. A functionality that may ease the usage of the software system for the user often increases internal complexity, which is not visible for the user [7]. *(4) Repeatability of similar elements* has a totally different character for software than it does for hardware. In software, similar elements exist just once as code, and then they are invoked where needed from all over the entire software system. Whether this increases or decreases the proportionality of complexity is hard to say. Nevertheless, it contributes heavily to intertwining and to an unclear graphical presentation of software.

Straight consequences. Brooks described in [8] many consequences from complexity. Consequences relevant in this context of interdisciplinary projects are depicted in Chapter 4.

4. Consequences for interdisciplinary projects

In this chapter, consequences from the essential differences between software and hardware are shown. The consequences can be classified to be associated with the combined product, the life cycle of the combined product, and the people involved in interdisciplinary projects. The list of consequences is most likely not complete. Nevertheless, it gives a clue regarding the severity of the consequences, which is sufficient for the understanding of the remainder of this paper.

4.1 Consequences associated with the combined product

In the following, consequences from the essential differences of software associated with combined products are explained.

Determination of the progress of completion. Due to the invisibility and intangibility of software, determination of the progress of completion of software is hard. An engineer in the automotive industry sees the progress of a new car model by how the prototype evolves and how the production plant for the mass production is built up. He has three indicators for the progress at hand. First, the material consumed, second, the visible growth, and third, the effort spent. Of course, in software development, effort spent is also an indicator, but a weak one. Effort is spent for what and for which things? The counterpart of consumed material could be the number or requirements implemented. Nevertheless, the determination of the number of requirements implemented depends, among other issues, on the chosen life cycle framework model (e.g., waterfall model, spiral model, or throwaway prototype model).

Presentation schemata are different. Due to invisibility, intangibility, and complexity, the presentation schemata of software and hardware are different. For the presentation scheme of hardware, the decomposition principle can be applied. For the presentation scheme of software, other principles such as abstraction [13] or global data storage might be superior. This difference makes it hard to represent a combined product in one schema.

Detection of errors: Due to the invisibility, intangibility, and the consequences of the non-applicability of natural constraints to software, the detection of errors is different in software and hardware. First of all, software does not have a link of form and function [13]. That means, a profound error is not as obvious as many errors in hardware. Beizer [7] identified three types of errors, differentiating software errors from hardware errors. That is, the space locality of a software error is manifested arbitrarily far away from the cause, the time locality of a software error is manifested arbitrarily long after the execution of the faulty code, and the consequences of a software bug are arbitrarily related to the cause. Another profound difference to hardware is that for software, no counterpart to continuous mathematics exists or is applicable to prove the reliability of complex software [16].

Duplication. Due to the intangibility, software is easy to duplicate. Duplication simply consists of copying code on another storage device. In contrast, in order to duplicate hardware, first a production plant is necessary, and second, the stages of the assembly line in the production plant must be run through.

The essential differences between software and hardware impose many severe consequences for combined products. The issues explained above show just a selection of consequences of combined products. Project staff on interdisciplinary projects must learn to handle these differences.

4.2 Consequences associated with the processes in the life cycle of a combined product

On a coarse level, software and hardware have the same processes in their life cycle, but with different characteristics. The characteristics vary with regard to implementation, cost, duration, effort, and accompanying activities such as quality assurance. The processes on the coarse level are the initial concept, the design, the production, the use, and the final disposal.

The initial concept consists of a feasibility study, market analysis, or similar activities. The design process in hardware can be divided into the processes of designing the prototype of a hardware model and the planning and build-up of the infrastructure of the production plant. For software, the design process is the development of the actual final product. In the production process, the hardware is duplicated multiple times in the production plant, whereas the production of software consists of simply copying. Use and final disposal are not important in this context.

The initial concept does not differ much for software and hardware. However, the design and production process reveal significant differences in implementation. With respect to cost, duration, and effort, the hardware production process costs more, is longer, and takes more effort than the hardware design process. On the other hand, the cost, duration, and effort in the software and hardware design process might be similar. With respect to the quality assurance activity, significant differences also exist. In software, quality assurance must be built in during the design phase, whereas for hardware, quality assurance starts with the production process.

The consequence for the processes and the accompanying activities in the life cycle of a combined product are manifold. In particular, special attention needs to be paid to the implementation of the accompanying activities.

4.3 Consequences associated with the people involved in interdisciplinary projects

People involved in the interdisciplinary projects are the customer, project management, software and hardware project staff. Apart from the software staff, it can be assumed that the others do have similar knowledge and attitude about software. Customer and project management still tend to be from the hardware side and therefore, their perception of software is often that software is easy to change and thus easy to be made conform. This attitude determines the procedure in the development of the combined product. They are not aware of what consequences arise from this perception. They are not aware that the determination of the progress of completion or the detection of errors is so different for software and hardware.

It can be assumed that especially the different perception of software among hardware and software people is one reason for the major problems in interdisciplinary projects.

5. A potential solution to cope with the differences and consequences: Interdisciplinary risk management

In order to cope with all the differences and consequences, means must be at hand that can help. Due to the diversity of the differences and consequences, a whole bundle of solutions might be necessary. On the other hand, the differences and consequences can be seen as risks in interdisciplinary projects. Risk is commonly defined as the possibility of suffering a loss [10].

One way to reduce or prevent the occurrence of risks, thus help projects to be successful, is the application of risk management. Considering the definition for risk management of PMBOK [17] and that for software project risk management of SEI [18], software risk management can be understood as a systematic, disciplined, continuous, proactive activity in which the processes of identifying, analyzing, responding to, monitoring and controlling of risks and their countermeasures are performed in order to maximize the probability and consequences of positive events and minimize the probability and consequences of events adverse to project objectives.

The benefits of risk management are that it supports the finding of items that can possibly harm the project, helps focusing on items that promise the most benefit, encourages proactive instead of reactive management, makes it possible to detect possible problems in early project phases (which will show up only in later phases without risk management), and supports the discovery and exploitation of opportunities.

What is special in interdisciplinary risk management is first, that it is especially designed to be applied in interdisciplinary projects and second, that project staff is familiar with one risk management process, terminology, and document templates. Such a uniform process would foster the communication and understanding among project staff. In the risk management sessions, which are performed regularly, all involved personnel will participate, that means representatives from all involved technical levels are present as well as from the different hierarchical levels.

Risk management is not a silver bullet. However, it can help to focus on the highest risks in the project and can find appropriate measures taking into account the objectives of the project.

In the following, the question is answered of how an interdisciplinary risk management approach can be achieved.

6. Achieving an interdisciplinary risk management approach

In order to achieve an interdisciplinary approach, three solutions are possible. The first solution is the development of a totally new risk management approach. The second solution is the adaptation of a software engineering risk management approach to hardware needs. And the third solution would be the adaptation of a hardware risk management approach to software needs.

The possibility of developing a totally new approach has the drawback that it must overcome the acceptance barrier. Furthermore, until such a new approach is matured, a lot of time goes by, which might not be available.

There are several reasons why we propose to adapt a hardware risk management approach to software needs and not vice versa. The first reason is that hardware risk management approaches are more commonly used in companies than software engineering approaches. A fact that can be explained due to many laws, norms, or directives that recommend or call for the application of a risk management approach for design and production processes. The second reason is that our experience shows that project staff is used to the "old" approaches. Now that they have to learn a lot about software engineering, they are reluctant to use or learn a new risk management approach as well. While they accept the need to change processes when they switch from mechanical to software engineering, they do not accept different terminology, forms, and processes for a supporting task like risk management. Therefore, to help such staff in upcoming interdisciplinary projects in which software plays a larger role, and to get a higher acceptance, we propose the adaptation of a hardware risk management approach.

In particular, we propose the adaptation of the Failure Modes and Effects Analysis (FMEA) as it is applied in the German automotive industry [5], [4] to the software needs.

Nevertheless, an important question for the midterm future is: Since it is obvious that software components increasingly permeate systems and software functionality becomes more and more complex, it might be more beneficial to adapt software risk management techniques to the needs of the other engineering disciplines. The hypothesis is that only in this way the challenges imposed by an increasing amount of software will be dealt with adequately. In the following chapter, an interdisciplinary risk management approach is described and results from a first application in an industrial setting.

7. Software Risk and Effects Analysis

The Software Risk and Effects Analysis (SREA) is based on the Failure Mode and Effects Analysis (FMEA). The idea was to create an approach suitable, first, for manufacturing companies that expand their products with software components and second, for companies already producing combined products, e.g., embedded systems. More information about SREA can be found in [19], [20], [21].

SREA paradigma. The SREA paradigm is not only an adaptation of FMEA to software needs, but also uses best practices to define a practice-oriented and effective risk management approach for interdisciplinary projects. The innovations encompass five aspects: (1) The explicit definition of a Risk Management Mandate process, (2) the adaptation of the three FMEA types to the specific requirements of the software development process, (3) introduced scalability with respect to the software-related needs of the project, (4) the inclusion of a software-adapted Ishikawa diagram, (5) new definition of the detection factor in the FMEA risk priority number (RPN).

SREA defines an explicit process for both product- and process-related risks. SREA product assesses the risks within the software itself (interfaces, modules, structure,…), whereas SREA process evaluates the software development process (requirements, frameworks used, testing). This separation enables continuous improvement of the development process, which also leads to a reduction of risks within every single product.

SREA Process Definition. SREA comprises three main processes. The first main process encompasses the risk management mandate and the goal adjustment process, which describe the communication to the stakeholders and the triggering of input artifacts of the risk management process. The incorporation of the stakeholder is transferred from Riskit [22] into SREA. The second main process, the analysis and structure process, visualizes and documents the project functions and their dependencies by running structural analysis, functional analysis, component clustering, net analysis, and risk identification. Furthermore, the third main process describes the evaluation of the risks and the actions set for risk reduction by running risk evaluation, risk control planning, and risk control.

The Net Analysis, in the second main process, is one of the most important procedures within SREA, especially for combined products. This procedure detects not only interfaces between manufacturing OR software parts of the system, but focuses particularly on the interfaces between manufacturing AND software parts. Customized Ishikawa diagrams for manufacturing and software engineering concern topics that are used to get an open minded view of product interfaces and risks.

Within the Risk Evaluation process, the risk priority number (RPN) is calculated by multiplying the factors of Severity, Occurrence, and Detection. The first factor is the severity number. It is an expression of the impact, in other words, the influence of a failure if it occurs. It depends on the influence on costumers, the functionality of the system, harm to the machine or to human beings, and the amount of work to solve the problem.

The second multiplier is the probability of occurrence. Development status, experiences as well as known failure scenarios of tested components are possible criteria for the estimation of occurrence.

Consequently, the third criterion for calculation is the probability of detection. In contrast to FMEA, this detection has to be adapted to the needs of software projects. The only possibility to detect a failure in a software component is to run tests. Therefore, the probability of detection gives information about the efficiency of the test method and, in addition, a ratio for possible detection of a failure before the software is delivered to customers.

Summarizing, the focus of SREA makes it possible to find risks that are not considered by traditional risk management methods, like FMEA. In addition to the strength of SREA in detecting risks within interfaces to other components, it has a second advantage by definition. On the one hand, it is possible to evaluate process-related risks. On the other hand, product-related risks can be evaluated. This enables evaluation of project risks from different points of view, leading to better results.

Case study results. The following section describes the environment and the design of the case study. Notice that this case study does not provide a full evaluation of the method, but was used as a first orientation for further experimentation and improvement.

The project in which SREA was evaluated was carried out at a manufacturing company in a traditional engineering field. The core business of the company is the production of cranes for international sales. Furthermore, the company is the world market leader in their field. They have developed a configuration software for their cranes that runs on web-based personal computers as well as on Personal Digital Assistants (PDA). Embedding the software on a small device such as a PDA with few storage resources forced them to evaluate the risks for the PDA application.

One of the main results of the case study was the discovery that the SREA approach identifies risks not only on the software level but also on the management, project, and context level. Additional risks within interfaces of the software and the environment were identified by SREA. Together with the scalability to certain components, this result could be a future advantage of the method.

According to the risk ranking, the main result was the identified influence of the detection factor. FMEA experts mentioned at meetings before the case study that this factor also causes some problems when FMEA is applied in its original context. In contrast, this study shows that there was an important influence on the result of the SREA risk evaluation process.

Regarding the effectiveness and efficiency of SREA, it can be said that, in principle, the method is applicable to combined products. For the exact evaluation of the method concerning effectiveness and efficiency, more experimentation has to be applied to get better results in the future.

In summary, several lessons learned resulted from the case study for the new SREA approach, on the one hand, and for Risk Management in general, on the other hand. Three of them are mentioned here. First, SREA was useful and applicable for the participants of the case study. Second, project members found risks that were not identified before despite continuous observation by the project team. Third, improvements with respect to experimentation for more convincing results were found.

8. Benefits for disciplines

As the number of mechanical, electrical, and software components constantly increases in many traditionally non-software products, and therefore more and more interdisciplinary projects are set up, the different disciplines must join forces in order to meet the requirements caused by the different attitudes of the disciplines. Project staff must learn to look beyond their own educational background and must accept that new ways of thinking are emerging. In other words, there is a need for project staff to solve problems in an interdisciplinary manner. The approach proposed here is meant to unify project staff with different backgrounds in order to perform successful projects. And if a good product is developed, then it is produced in a successful engineering project. Consequently, not only does the discipline of software engineering benefit from this experience, but so do the other engineering disciplines.

9. Summary

This paper is the extension of a position paper presented at the STEP2003 Second Workshop for Interdisciplinary Risk Management. This series of workshops intends to identify techniques successfully applied to disciplines cognate to software engineering. The participants of the workshop wanted to know to which extent these techniques can be transferred to software engineering or what can be learned for software engineering. In this sense, this paper proposes the adaptation of an existing risk management approaches originally developed for a single discipline to the needs of interdisciplinary projects.

9. References

[1] S. Hartkopf, P. Kaiser, and B. Freimut. Interdisciplinary risk management - A position paper. Technical Report IESE-Report No. 091.03/E, Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany, Sauerwiesen 6, D-67661 Kaiserslautern, 2003. http://www.iese.fraunhofer.de/pdf_files/iese-091_03.pdf.

[2] P. Brereton, N. Mehandjiev, and P. Layzell. Interdisciplinary software engineering. In F. Coallier, G. Hoffnagle, P. Layzell, L. O´Brien, and D. Poo, editors, Proceedings of the Tenth International Workshop Software Technology and Engineering Practice (STEP), page 45. IEEE Computer Society, Los Alamitos, California, USA, October 2002.

[3] N. Mehandjiev, P. Layzell, P. Brereton, G. Lewis, M. Mannion, and F. Coallier. Thirteen knights and the seven-headed dragon: an interdisciplinary software engineering framework. In F. Coallier, G. Hoffnagle, P. Layzell, L. O'Brien, and D. Poo, editors, Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP 2002), pages 46-54. IEEE Computer Society, 2002.

[4] Arbeitsgruppe 131 "FMEA" Deutsche Gesellschaft für Qualität e.V., Frankfurt am Main, Germany. FMEA - Fehlermöglichkeits- und Einflussanalyse. Beuth, 2 edition, 2001.

[5] Verband der Automobilindustrie e.V. (VDA). Qualitätsmanagement in der Automobilindustrie: Sicherung der Qualität vor Serieneinsatz Teil 2, System-FMEA. Verband der Automobilindustrie e.V. (VDA), Qualitätsmanagement Center (QMC), Karl-Hermann-Flach-Str. 2, 641440 Oberursel, Germany, Fax: +49 (0)6171 9122-14, 1st edition, 1996.

[6] O. Mäckel. Mit Blick aufs Risiko - Software-FMEA im Entwicklungsprozess softwareintensiver technischer Systeme. QZ Qualität und Zuverlässigkeit: Qualitätsmanagement in Industrie und Dienstleistung, 46(1):65-68, 2001, (in German).

[7] B. Beizer. Software is different. Annals of Software Engineering, 10(1-4):293-310, 2000.

[8] F. P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. Computer, 20(4):10-19, 1987.

[9] anonymous. Collins English dictionary. HarperCollins Publishers, PO Box, Glasgow G4 0NB, 3rd edition, 1991.

[10] anonymous. The American Heritage Dictionary of the English Language. Third Edition. Houghton Mifflin Company, 3rd edition, 1996.

[11] M. Glinz. Skript zur Software Engineering I Vorlesung im Wintersemester 2002/2003 an der Universität Zürich, Institut für Informatik. available on July 10, 2003 on http://www.ifi.unizh.ch/groups/req/ftp/se_I/kapitel_01.pdf, 2002.

[12] M. Broy and D. Rombach. Software Engineering: Wurzeln, Stand und Perspektiven. Informatik Spektrum, 16:438-451, Dezember 2002.

[13] G. A. King. Quality technique transfer: Manufacturing and software. Annals of Software Engineering, 10(1-4):359-372, 2000.

[14] N. G. Leveson. Safeware: system safety and computers. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[15] IEEE Computer Society, IEEE Standard Glossary of Software Engineering Terminology, Los Alamitos, California, USA: IEEE Computer Society, 2003.

[16] D. L. Parnas. Software aspects of strategic defense systems. Communications of the ACM, 28(12):1326-1335, December 1985.

[17] W. R. Duncan. A Guide to the Project Management Body of Knowledge. PMI Project Management Institute, Newtown Square, PA, USA, 1996.

[18] C. J. Alberts, A. J. Dorofee, R. P. Higuera, R. L. Murphy, J. A. Walker, and R. C. Williams. Continuous Risk Management Guidebook. SEI, Carnegie Mellon University, Pittsburgh, 1989.

[19] T. Kurz. Risk Management for Embedded Systems: The Software Risk and Effects Analysis (SREA). Master's thesis, School of Telecommunications Engineering of the University of Applied Sciences and Technologies, June 2003.

[20] S. Hartkopf, T. Kurz, and T. Heistracher. Risk management as a means to better manage interdisciplinary projects. In A. Fricke, G. Kerber, D. Lange, and R. Marre, editors, Proceedings of the second conference on Interdisciplinary Project Management 2004 (InterPM2004), pages 259-273, GPM Deutsche Gesellschaft für Projektmanagement e.V., Roritzerstr. 27, 90419 Nürnberg, Germany, March 2004.

[21] S. Hartkopf, T. Kurz, and T. Heistracher. Risk management as a means to better manage interdisciplinary projects. Technical Report IESE-Report 045.04/E, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 2004.

[22] J. Kontio. The Riskit method for software risk management,version 1.00. Computer Science Technical Reports CS-TR-3782, University of Maryland. College Park, MD, 1997.