Mälardalen University Licentiate Thesis
No.107

Exploring Sustainable Industrial Software System Development

within the Software Architecture Environment

Pia Stoll

October 2009

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 978-91-86135-36-2
Printed by Mälardalen University, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

This thesis describes how sustainable development definitions can be transposed to the software architecture environment for the industrial software system domain. In a case study, sustainable development concerns from three companies are investigated for their influence on the dimensions of sustainable development: economical, environmental, and social sustainability. Classifying the case study's concerns, in the thesis's Software Engineering taxonomy, shows that the software development concerns are in majority and the software architecture concerns surprisingly few. The economical sustainability concerns dominate followed by social sustainability concerns, including both concerns successfully met and concerns to be met.

Sustainable industrial software system development is in the thesis defined as: "Sustainable industrial software system development meets current stakeholders' needs without compromising the software development organization's ability to meet the needs of future stakeholders". Understanding current and future stakeholders concerns is necessary for the formulation of sustainability goals and metrics. Clarifying the interrelationships among stakeholders' concerns' impact on business goals and software qualities, in the thesis's Influencing Factors method, proves to help stakeholders understand their future needs.

Trust is found to be critical for sustainable development. For the establishing of trust between system and system users, the usability quality is vital. To implement usability support in the architecture in the early design phase, reusable architectural responsibilities are created. The reusable architectural responsibilities are integrated into an experience factory and used by the product line system architects, resulting in a return of investment of 25:2.

Swedish Summary - Svensk Sammanfattning

Avhandlingen beskriver hur definitioner av hållbar utveckling kan översättas till mjukvaruarkitekturens omgivning för industriella mjukvarusystem. Systemintressen, avseende hållbar utveckling, samlas in i en intervjustudie och relateras till dimensionerna: ekonomisk hållbarhet, social hållbarhet, och miljömässig hållbarhet. En taxonomi av arkitekturbeskrivningar skapas, som införlivar både "Enterprise Architecture" och "Software Engineering" beskrivningar. Klassificerade intressen visar på stort fokus på verksamhetsaspekter och överraskande lite fokus på mjukvaruarkitektur. En naturlig följd är att ekonomisk hållbarhet står i centrum, följd av social hållbarhet.

Hållbar utveckling av industriella mjukvarusystem definieras i avhandlingen som: "Hållbar utveckling av industriella mjukvarusystem tillgodoser systemets nuvarande intressenters behov utan att äventyra organisationens möjligheter att tillgodose framtida intressenters behov". Klargörande av inbördes förhållanden mellan intressen och deras påverkan på affärsmål och mjukvarukvalitet hjälper organisationen att införliva intressenters behov med strategin för hållbar utveckling. Den i avhandlingen konstruerade "Influencing Factors" metoden tydliggör relationerna mellan systemintressen i en fallstudie, vilket visar sig hjälpa studiens intressenter förstå sina framtida behov.

Tillit är en viktig del i hållbar utveckling. För att skapa tillit mellan systemet och dess omgivning, är användbarheten viktig. Arkitekturella mönster med användbarhetsstödjande instruktioner tillämpas i avhandlingen. Fyra systemrelaterade generella funktioner identifieras, med arkitekturella instruktioner för att garantera funktionernas användbarhet. Kunskapen görs tillgänglig för studiens produktlinjearkitekter i en "erfarenhetsfabrik". Resultatet är en rapporterad kostnadsbesparing av 25:2.

To Alex, Simon, and Sofie

Preface

The challenges have been many during the writing of this thesis and I'm forever grateful to many of you out there who have served as an inspiration and guidance.

I would especially like to thank my supervisors, Christer Norström and Anders Wall, for structuring my contributions and for trying to understand my reasoning during these years even if the topic have been slightly off the one of the department's regular thesis. Someone who has supported many of my ideas is my present group manager, Magnus Larsson. Another group manager of mine, Fredrik Ekdahl was also a great support and enabler for my PhD studies.

The project, where the ideas of the Usability Supporting Architecture Pattern were tested, included two persons at ABB Force Measurement who have been very supporting throughout the USAP project and therefore contributed to the USAP project's success leading to some major contributions to my thesis. Thank you Fredrik Norlund and Jan Hasselgren. In the same project, I would like to direct a thanks to Bonnie E. John, Len Bass, and Elspeth Golden. We have had a handful of very intense and stimulating discussions and I have always left them revived and full of new ideas. I also wish to thank the BESS group at MDH for all of the interesting discussions around business and software engineering. For uplifting discussions around everything but software engineering: thank you Sara, Shiva, Helena, Ambra and Åsa.

My beloved family: my husband Alex and my children Simon and Sofie, my mother, my brothers Patrik and Niclas with families and my friend Hanna Hagmark Cooper; thank you for being there!

Pia Stoll
Västerås, September 15. 2009

List of included Publications

The content of this thesis has been published in the following papers. References to the papers will be made using the alphabetic association of the papers.

A. Guiding Architectural Decisions with the Influencing Factors Method, Pia Stoll, Anders Wall, Christer Norström, Working IEEE/IFIP Conference on Software Architecture (WICSA), Vancouver, BC, Canada, February, 2008

B. Achieving Sustainable Business for Industrial Software Systems, Pia Stoll, Anders Wall, Business Sustainability Conference, Ofir, Portugal, June, 2008

C. Preparing Usability Supporting Architectural Patterns for Industrial Use, Pia Stoll, Len Bass, Bonnie E. John, Elspeth Golden, International Workshop on the Interplay between Usability Evaluation and Software Development, I-USED, CEUR Workshop proceedings series, ISSN 1613-0073, Pisa, Italy, September, 2008

D. Supporting Usability in Product Line Architectures, Pia Stoll, Len Bass, Bonnie E. John, Elspeth Golden, Software Product Lines Conference, SPLC, San Francisco, USA, August, 2009.

E. Software Engineering featuring the Zachman Framework, Pia Stoll, Anders Wall, Christer Norström, Technical Report, ISSN 1404-3041 ISRN MDH-MRTC-240/2009-1-SE, School of Innovation, Design and Engineering (IDT), Mälardalen University, Sweden, 2009.

F. Applying the Software Engineering Taxonomy, Pia Stoll, Anders Wall, Christer Norström, Technical Report, ISSN 1404-3041 ISRN MDH-MRTC-241/2009-1-SE, School of Innovation, Design and Engineering (IDT), Mälardalen University, Sweden, 2009.

Additional Publications

- A Responsibility-Based Pattern Language for Usability-Supporting Architectural Patterns, Bonnie E. John, Len Bass, Elspeth Golden, Pia Stoll, ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS, Pittsburgh, USA, June, 2009
- Usability Supporting Architecture Pattern for Industry, Pia Stoll, Fredrik Alfredsson, Sara Lövemark, Industrial Experience Report, Nordic Computer Human Interaction Conference, NordiCHI, Lund, Sweden, 2008
- Reconstructing the Architecture Model for a Sustainable Software System, Pia Stoll, Industrial Experience Report, SEI Architecture Technology User Network, SATURN, Pittsburgh, USA, 2008
- Identifying Sustainable Systems Architecture's Primary Concerns, Roland Weiss, Pia Stoll, Industrial Experience Report, SEI Architecture Technology User Network, SATURN, Pittsburgh, USA, 2008
- Integrating Usability Supporting Architectural Patterns in a Product Line System's Architecture, Pia Stoll, Len Bass, Bonnie E. John, Elspeth Golden, Industrial Experience Report, SEI Architecture Technology User Network, SATURN, Pittsburgh, USA, 2009

Contents

I

Thesis

Chapter 1

Introduction

Industrial software systems are in this thesis the synonym for complex control and supervision systems used in power and automation utilities and plants of various art. Not long ago these systems used a rather small amount of software. But this has changed and today the systems have a relatively high degree of software and are almost autonomous. The role of the operators has shifted from having to use their expertise to set the correct control values manually to a role of supervision and fault finding. One system can nowadays control a plant without any operator interaction and the system interfaces with a multitude of external systems. Software complexity has grown in the same pace as the system's amount of software has increased. When the features that once were performed by hardware now are replaced by software, the software parts can interact with each other in a way the hardware parts could not. This is used to create additional value. Industrial software systems are getting more and more sophisticated. Customers are offered more and more features.

As the offering increases, yesterday's advanced features are turning into commodity. To get a return of investment for both customers and development organization, the system has to be maintained and stay operational for decades, i.e. the system has to become sustainable.

Pollan has defined an unsustainable system simply as *"a practice or process that can't go on indefinitely because it is destroying the very conditions on which it depends"* [1]. Unruh has argued that numerous barriers to sustainability arise because today's technological systems were designed and built for permanence and reliability, not change [2].

"A global agenda for change" - was what Gro Harem Brundtland, as the

chairman of the World Commission on Environment and Development, was asked to formulate in 1987 [3]. As a result, the Brundtland commission defined sustainable development as:

> *Sustainable development is development that meets the needs of the present without compromising the ability of future generations to meet their own needs. It contains within it two key concepts: the concept of "needs", in particular the essential needs of the world's poor, to which overriding priority should be given; and the idea of limitations imposed by the state of technology and social organization on the environment's ability to meet present and future needs.*

In [4], Dyllick and Hockerts transpose the definition to the business level:

> *Corporate sustainability is meeting the needs of a firm's direct and indirect stakeholders (such as shareholders, employees, clients, pressure groups, communities etc), without compromising its ability to meet the needs of future stakeholders as well.*

Following the reasoning of the Brundtland commission [3] and Dyllick and Hockerts [4], sustainable industrial software development would be defined as:

> *Sustainable industrial software development meets the needs of the software development organization's direct and indirect stakeholders (such as shareholders, employees, customers, engineers etc), without compromising the organization's ability to meet its future stakeholders' needs as well.*

In this thesis, the term "Corporate Sustainability" is used when the work referred to uses the term. Otherwise the term "Sustainable development" is used.

Three dimensions of corporate sustainability are outlined by Dyllick and Hockerts: Environmental sustainability, Economic sustainability, and Social sustainability, the "triple-bottom-line" in Figure 1. Dyllick and Hockerts conclude that a single-minded focus on economic sustainability can succeed in the short-run; however, in the long-run sustainability requires all three dimensions to be satisfied simultaneously.

Sustainable development of industrial software systems is a true challenge due to changes in concerns originating from: new technology, new stakeholder

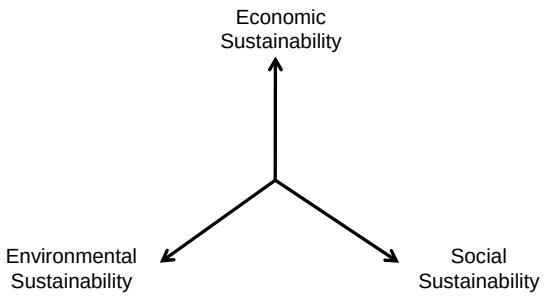

Figure 1: Three dimensions of corporate sustainability

needs, new organizations, and new business goals during decades. It's challenging since it has not been researched for industrial software systems and the domain need an understanding of the success-critical concerns related to the achievement of sustainable development of systems as the complexity of organizations, processes, and architectures increase.

Organizational complexity involves many success-critical stakeholders, often located all over the world, who have to reach a consensus around the most important business goals for the system now and in the next future. Sustainable systems has built-in legacy heritage to consider as well as present software architecture and design when introducing new business goals. If the organization in the past predicted today's stakeholders' needs and adapted the past development to today's predicted needs, the incorporation of today's concerns in the system should be fairly straightforward. In the same fashion, today's organization needs to predict future stakeholders' needs and select the most valuable concerns to address. To do this, the architects need an understanding of how the stakeholders' concerns affect business goals and architectural qualities. For example, industrial software systems are often affected by company mergers and acquisitions, where two or more systems have to be consolidated into one system or the systems have to share a core part. The effect of such decision on software quality is hard to overlook. Sustainability is therefore related not only to software structures and their interactions but also to the system's environment in terms of the enterprise aspects as organization, business, tactics and scope. Enterprise aspects have not been put in relation to software architecture and implementation for industrial software systems in an explicit way earlier. As organizational complexity grows, the impact of the enterprise aspects on

the software system is significant.

Systems not being usable will not be sustainable in the future. Releasing a system with usability problems is decreasing the trust between users and system, thereby decreasing the economical sustainability. Industrial software systems must find a way of implementing usability support early in the development phase since the development phase of an industrial software system is likely two years or more. Redoing two years of architectural design and implementation due to usability problems is not an option. But traditional usability engineering suggests usability tests with working functionality when a prototype is at hand which is late in the design phase. From the usability tests, list of usability flaws go back to the developers. Problems related to the user interface's design can often be fixed but problems requiring architectural refactoring are too expensive to correct. Correcting the architecture related problems would also cause an unacceptable delay in the release date. This is especially critical when developing product line systems. Product line development typically develops the core architecture and its implementation first. Then each product's specialization is developed and first after that has been done, the product's usability can be fully tested.

1.1 Research Rationale

Sustainable development is defined in terms of meeting current stakeholders' needs without compromising the software development organization's ability to meet the needs of future stakeholders. For software development, the definition raises the question: how can stakeholders be encouraged to contribute with their concerns and how can their concerns' impact on the software development be clarified? If the stakeholders concerns are not understood, they can hardly be met. But at the same time the system's environment is getting more and more complex and the number of stakeholders increases. One system can hardly meet all concerns. The concerns can not be prioritized for an increase in sustainability if their impact is not known. Stakeholder participation is a social sustainability criteria according to Labuschagne et al. [5]. In software engineering, methods like the Quality Attribute Workshop [6] engage stakeholders to participate and voice current concerns. The Architecture Trade-Off Analysis method [7][8] is another method that stimulates success-critical stakeholders to voice concerns regarding the system development. Inviting stakeholders to workshops as the Quality Attribute Workshop [6] and the Architecture Trade-Off Analysis Method [7] are two ways, but may need a complement due to the

scalability issue. The number of stakeholders and their number of locations increase as the distribution of development and management increases. The chance of gathering a large set of distributed stakeholders for a one-day QAW or a five day ATAM workshop on a continuous basis is very small in the domain of industrial software systems where no standards or regulations enforce these kinds of workshops. Stakeholders' concerns change frequently and the analysis of the concerns must be updated just as often not to miss an important concern that need to be met in order to maintain sustainable development. Another aspect, highly relevant for sustainable development, is the needs of the future stakeholders. The QAW and the ATAM gather current stakeholders and extract their needs by analyzing their voiced concerns. Experiences from three Quality Attribute Workshops show that stakeholders have a very strong urge to voice concerns related to their own working environment and hardly ever voice a concern not related to themselves. To achieve sustainable development, the company must predict future stakeholders' needs by analyzing the future stakeholders' concerns. An *on/off-line, light-weight stakeholder concern collection, prediction and analysis method* is therefore needed that could include concerns from future stakeholders.

Context will be very important when defining what sustainable development means to the stakeholders. Following the conclusions of Reed [9], Salzmann et al. [10], and Sing et al. [11], each community or domain should: interpret corporate sustainability, argue its business case for corporate sustainability, and establish their own corporate sustainability assessment. For the domain of industrial software systems, this translates into a need of a *case study that explores the industrial software system domain's stakeholders' sustainable development concerns* in order to find the most important sustainable development goals. Finally, metrics should be established to assess the processes of achieving the goals. Very little research has explored the industrial software system domain to find the scope, business case and metrics for sustainable development.

Research in the area of sustainable development for software systems will be extra challenging since it involves aspects from: enterprise architecture, economical theory, organization theory, software engineering and cognitive psychology. There exists no framework where views describing all these aspects can be investigated for their interrelationships. The aspects are part of different research disciplines. However, enterprise architecture and software engineering are closely related. In traditional software architecture, a component may be a procedure, a process or an object-oriented object [12][13]. The enterprise software system is a "system of systems" in the sense that the components of

the enterprise system are normally considered as systems in the (developer-oriented) traditional software architecture [14]. Zachman has set up a framework for describing system information of a complex object from different usage perspectives, and from the journalistic context abstractions: "What", "How", "Where", "Who", "When", and "Why" [15][16][17][18]. Initially, he described the framework by collecting data from the building engineering domain, applied the framework to the data from the aircraft engineering domain and finally applied the framework to the enterprise systems domain. Software engineering has also been inspired in much of its research from the building engineering domain. Especially the software engineering pattern community [19][20][21][22] has used concepts from the pattern language theory of the building architecture researcher Alexander [23][24]. There would be a benefit of classifying software engineering concepts into the Zachman framework to find out if the framework can accommodate all the concepts, and if so, to find out how software engineering relates to the enterprise views for the system's operational and development environment. The result would be *a derivative of the Zachman framework for software engineering* that could classify sustainable software system development concerns related to the system's environment in term of scope, business, system, software, and components.

A crucial ingredient in social sustainability is trust. Trust among employees is the prerequisite for social capital enforcements in networking, knowledge sharing, commitment etc. Trust in the relation between customers and the software system development company is achieved by the system having a certain set of qualities. Hoffmann et al. describes a trust model that goes beyond security [25]. The trust model includes: reliability, safety, security, availability, privacy, user expectations, and usability. Human trust in automation is translated into trust as the expectation that a service will be provided or a commitment will be fulfilled.

Trust in the relation between customers, as external stakeholders, and the system increases economic sustainability. Usability is an important factor in the trust between system customers and the system. But *usability support in a system's software architecture* has been shown to be very superficially described, mostly as a separation of concerns between user interface logic and the rest of the system's logic [26][27][28][29][30][31]. Usability problems are usually discovered after the product's release when the architecture no longer can accommodate the problems' solutions. If usability support in the architecture could be built in early, the economic sustainability would increase also in terms of potential financial profit, by speculating in an increased sales by offering more usable systems. The reputation of the system would also increase,

which increases the economic sustainability. The research by Folmer [28][29] deals with evaluation of architecture for usability support. The research by Juristo et al. [30][31][32] describes usability issues with a possible impact on software design as usability patterns, but does not suggest any way of designing the architecture to support the usability issues. The work of Bass et al. [26][27] does suggest a way of designing architecture to support success-critical usage scenarios in the form of Usability-Supporting Architecture Patterns.

1.2 Research Questions

Considering the lack of usability tactics to apply in software architecture to avoid unsolvable usability problems and to reinforce trust between system and system users, the following research question is formulated:

RQ1 "How can support for usability be built into software architecture of industrial software system in the early design phase?"

To be able to understand the success-critical concerns adding most value to the goal of sustainable development, the following research question is formulated

RQ2 "What are the concerns affecting the sustainable development of an industrial software system?"

Sustainable development is depending on the knowledge of current and future stakeholders' needs in order to meet those needs. Considering the importance of the explicit knowledge of current stakeholders' needs and future stakeholders' needs and the impact of these needs on business goals and software qualities, the following research question is formulated:

RQ3 "How can current and future stakeholder concerns be collected and analyzed for their impact on business goals and quality attributes in the domain of industrial software systems?"

Sustainable industrial software system development concerns will have aspects concerning the economical sustainability, social sustainability, and environmental sustainability. The aspects will relate to: enterprise architecture, economical theory, organization theory, software engineering and cognitive psychology. To find out how software engineering relates to the enterprise views for the system's operational and development environment, the following research question is formulated:

RQ4 "How can industrial software system stakeholders' concerns be described by views in an enterprise framework, that incorporates software engineering artifacts descriptions?"

1.3 Thesis Outline

Chapter 2 describes the work relating sustainable industrial software system development and: software engineering, software architecture, enterprise architecture, usability, software development measures, and software quality attributes. In chapter 3, the research design of this thesis is described. The contributions of this thesis are described in chapter 4. Finally, future work is presented in chapter 5.

Chapter 2

Related Work

The following sections describe the software architecture environment and its relation to the economical, social, and environmental sustainability dimensions.

2.1 Sustainable Development and Software Engineering

Corporate sustainability implies a much broader interpretation of the concept of capital than is used normally by either economists or ecologists. Economic, natural, and social capital each have different properties and thus require different approaches.

Economic sustainability requires firms to manage three types of economic capital:

1. Financial capital, i.e. equity and debt
2. Tangible capital, i.e. machinery, land and stocks
3. Intangible capital, i.e. intellectual property, internal systems, methods, tools, external customer loyalty and brand

The examples of intangible capital are from Sveiby's framework for categorizing and measuring the intangible assets [33].

The third line of Dyllick's three corporate sustainability dimensions is the environmental sustainability, Figure 1. Ayres argues that if the industrial organism consumes more energy and materials than can be reproduced, or if it emits more emissions than can be absorbed through natural sinks, the industrial system becomes ecologically unsustainable [34]. Dyllick's definition of environmental sustainable systems says that such systems do not engage in activity that degrades eco-system services (i.e. climate stabilization, water purification, soil remediation, reproduction of plants and animals) [4]. Systems that enable utility and industry customers to improve their performance while lowering environmental impact should therefore contribute to natural resources being consumed in a lower pace, even if the systems themselves do not increase natural resources. If the systems are consuming less natural resources in their development and operation than they help utilities and industry customers to save, then the systems should be contributing to the environmental sustainability according to the definitions by Ayres [34] and Dyllick [4]. Environmental sustainability is impacted by the software system's structures and inter-operations. Google develops software that consumes huge amounts of natural energy resources. Every time someone taps a Google search button, thousands of servers are activated. One of Google's server plants can be expected to demand the same amount of energy that could power 82,000 homes [35]. As a response to this issue, Google invests tens of millions dollar in research and development in renewable energy.

Corporations are the fundamental cells of modern economic life according to Dunphy, Griffiths and Benn [36]. They state that "Corporation not sustaining will not be sustainable". Also software systems consume energy and if they don't sustain by building or retaining natural capital, they will not be sustainable.

Social sustainability is defined in relation to human capital and society capital. Human capital is represented by e.g. skills, motivation, and loyalty of employees and business partners. Society capital is e.g. quality of public services. Coleman introduced a conceptual tool which he called "social capital" in 1988 [37]. Social capital, according to Coleman, is increased by social networks where trustworthiness is an important factor. Coleman shares the view on human capital with Dyllick and Hockers [4] and describes human capital as being created by changes in persons that bring about skills and capabilities that make them able to act in new ways. Human capital among employees is strengthened if the managers take an interest in strengthen their own human capital in order to support the employees in their education.

Information channels are an important form of social capital according to

Dyllick [4]. Technology interested employees who on their own initiative find out current technology trends and discuss these with managers and coworkers, save the company the time of paying an employee to do technology scouting. A different value to the social capital arise when there are key-competences in an organization to whom others turn for help. The key-competence, helping a coworker, trusts the coworker to return the favor in the future, which establishes an obligation on the part of the coworker. Shifting development from an organization, that relies on key-competences, to a low-cost country in order to save development cost translates into a shift between social capital and economic capital. The coworkers in the remotely located organization have no direct access to the social capital of the key-competences. The value of the social capital of the key-competences decreases since it can not easily be accessed, but the economic capital is strengthened by the savings in employees' salaries.

Figure 2 shows the concept of corporate sustainability from the perspective of added stakeholder value. If the company ignores one dimension of the sustainability, e.g. environmental sustainability, in order to maximize added value to the current stakeholders, then the added value to the future stakeholders likely will be reduced. The car industry is one example of this. For long times the car industry ignored the future stakeholders' need of a healthy environment and produced cars consuming too much of the nature's energy resources. If the industry would continue to produce cars this way, the car industry would not have sustainable development. Current stakeholders demands for less energy consuming cars has contributed to the car industry's rethinking making it increase its environmental sustainability. Current customers get an added value by taking on the responsibility of preserving added value to the future customers. Additionally to the customers taking on this responsibility, environmental regulations are forced upon the car industry by the political sphere.

O'Connor discusses the interfaces between the three dimensions of corporate sustainability [38]. A new concept of "spheres" is used, replacing the sustainability dimensions. A forth sphere is added, the political sphere, that should regulate the economic sphere's relation to the other spheres. The forth sphere takes on the responsibility of ensuring added value to the future stakeholders.

Motivating sustainable development, i.e. creating a business case for sustainable development, is not obvious considering all dimensions of sustainability. Reed examines the business case for corporate sustainability strategies and does an attempt to quantify it financially [9]. Shareholder value is in focus and the financial case is made at company level in the context of that company's

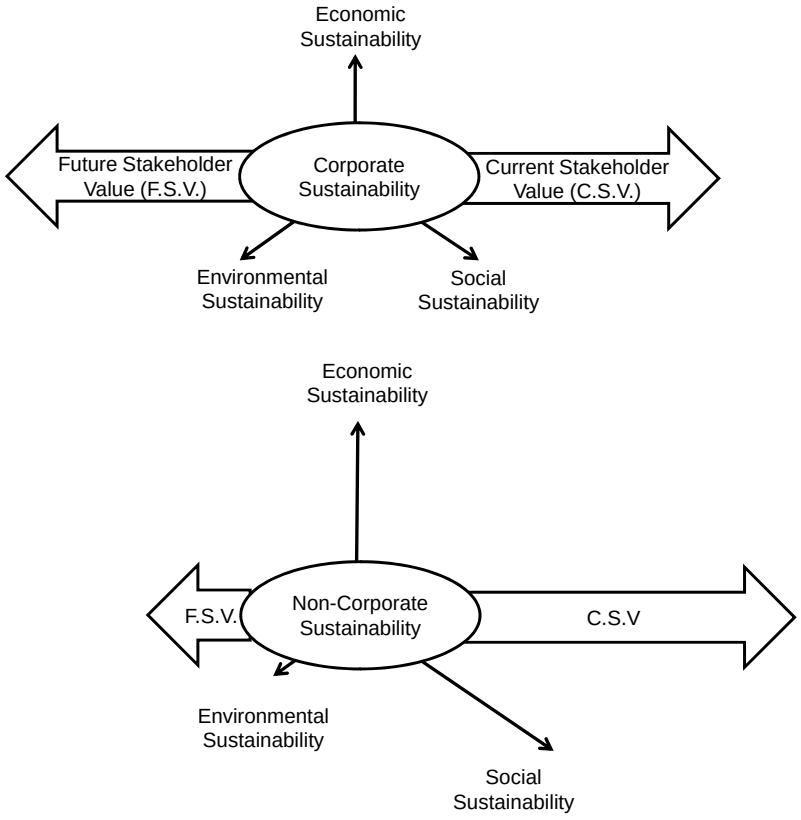

Figure 2: *Top figure*: The company's corporate sustainability is balancing added value to its current stakeholders with added value to its future stakeholders. *Bottom figure*: The company's non-corporate sustainability is maximizing added value to its current stakeholders, by ignoring environmental sustainability, thereby reducing the added value to the future stakeholders.

strategy within their industry. Sustainable development for the industrial software system domain will, following Reed's reasoning, have another business case than e.g. the chemical engineering domain. One of Reed's conclusion is that:

> *. . . it is up to those companies that believe they are creating value through sustainability strategies to clearly articulate that value to investors and financial analysts.*

Salzmann, Ioenescu-Somers and Steger identify insufficient understanding of managers' key arguments for corporate sustainability [10]. They attribute this primarily to lack of descriptive research in the areas of: how business cases are built; and how effective they are and what barriers they face. Salzmann concludes that research must identify managers' key economic arguments used to drive corporate sustainability internally in the company and the success-factors of these arguments. Most likely the business case is often based on the enforced regulations by the "forth sphere", the political sphere, on the firm. If the firm does not conform to the regulations it will not be allowed to sell its products or face legal consequences. However, most companies use the enforced regulations to market themselves as environmentally friendly, thereby increasing the economical sustainability by increasing the environmental sustainability.

When the dimensions and the motivation of sustainable development and their applicability to a specific company have been understood, the company's interest will be in establishing sustainability criteria. Labuschagne et al. propose a comprehensive framework of sustainability criteria that can be used to assess the sustainability of projects, technologies, as well as the overall company sustainability [5]. The sustainability criteria framework considers the dimensions: environmental sustainability, economic sustainability, and social sustainability. Economic sustainability is, besides common accepted criteria such as Net Present Value (NPV) and Return Of Investment (ROI), suggested to be measured by the criteria: potential financial benefits.

In the "Framework for sustainability assessment tools" article [39], Ness et al. categorizes sustainability assessment tools in three general areas: indices/ indicators; product-related assessment tools; and integrated assessment. Life cycle management is an assessment tool in the product-related assessment tool area. Risk analysis and uncertainty analysis are part of the integrated assessment tool area. Sing et al. have made an comprehensive overview of sustainability assessment methodologies [11]. Their conclusion is that various international efforts on measuring sustainability exist, but only few of them have an integral approach taking into account environmental, economic and social

aspects. As sustainable development is about the inter-linkage of the three aspects, trying to use the efforts supplementary will be missing the point of sustainable development. Finally, Sing et al. state that sustainability assessment techniques should be selected and negotiated by the appropriate communities in interest.

Following the conclusions of Reed [9], Salzmann et al. [10], and Sing et al. [11], each community or domain should: interpret sustainable development, argue its business case for sustainable development, and establish their own sustainable development assessment.

Possible criteria for the software engineering domain that fits the descriptions of sustainable development by [4][37][5] are listed in Figure 3. If these are actually criteria used or criteria that should be used in industrial software system development organizations is an open issue that needs further research. It's not known how the business processes of the software development organization relate to the sustainability criteria.

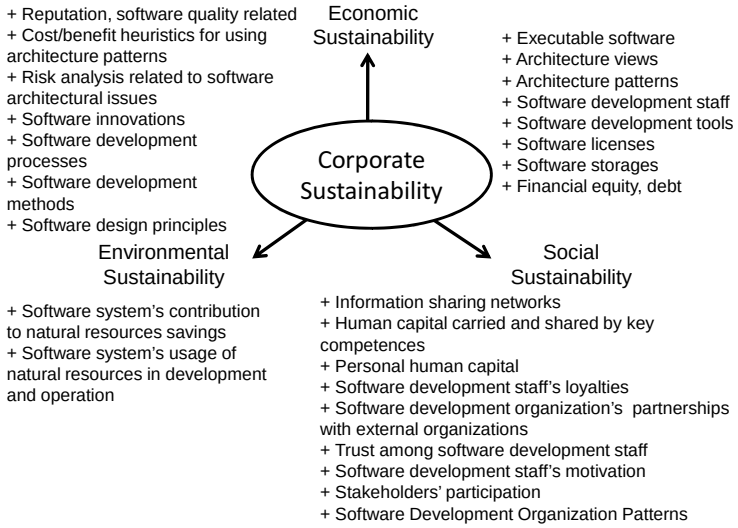

Figure 3: Three dimensions of corporate sustainability with possible criteria from the software engineering domain and the software architecture's environment

Another open issue is how the software quality concerns relate to sustain-

ability criteria. Ozkaya et al. make the case that software architectural patterns carry economic value in the form of real options, providing designers with the right, but not the obligation, to take subsequent design actions in the future in the face of uncertainty [40]. Their method could be one way of measuring potential financial benefits for a software development organization as a part of measuring economic sustainability. The stakeholder value that is connected to sustainability criteria, i.e. which stakeholder will perceive an added-value when the criteria is reached, should be explored.

Basili, Cladiera, and Rombach describe the Goal Questions Metric approach [41]. By understanding specific aspects of a concern, related to the system development, the goal can be set and metrics to achieve the goal constructed. For sustainable development this would translate into collecting stakeholders' concerns regarding the system and its development and usage. Then the aspects of those concerns related to sustainable development have to be understood and goals and metrics defined.

Jain and Boehm describe an initial theory of value-based software engineering [42]. The value-based software engineering (VBSE) theory addresses the questions of "which values are important?" and "how is success assured?" for a given software engineering enterprise. Their theory could be used to find out, what values are important to achieve success in form of sustainable development. Assigning a sustainable development value to each one of the current and future stakeholders' concerns, can aid in the elicitation of concerns important to address for the achievement of sustainable development.

2.2 Software Engineering

The term software engineering first appeared in the 1968 NATO Software Engineering Conference[1] and was meant to provoke thought regarding the current "software crisis" at the time.

In the article "Will There Ever Be Software Engineering" [43], Jackson claims that:

> *. . . there will never be software engineering. As these specializations flourish (e.g. compiler engineering, operating systems [author's remark]) they leave software engineering behind . . . A professor of software engineering must, by definition, be a professor of unsolved problems.*

[1] http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html

Ziv and Richardson state the uncertainty principle of software engineering (UPSE) [44] as:

> *Uncertainty is inherent and inevitable in software development processes and products.*

They describe the software development as a complex human enterprise carried out in problem domains and under circumstances that are often uncertain, vague or otherwise incomplete.

Basili and Musa discuss the management perspective of software engineering [45]. To understand where the time and effort in software development are going, Basili and Musa suggest that:

> *. . . we must isolate and categorize the components of the software engineering discipline, define notations for representing them and specify the interrelationships among them as they are manipulated.*

They point to a set of areas that they believe will play an important role in deepening the understanding and attainment of software quality. These areas are:

- Formal methods
- Design methods, e.g. object oriented design
- Measurement approaches
- Usage and reduced-operation software
- Reuse
- Cognitive psychology, e.g. problem solving research
- Software sociology, e.g. group dynamics, communication networks, and organizational politics

Where Jackson sees the application domain of software engineering as crucial for the science of software engineering, Basili and Musa see the software development related activities, applicable to any domain, as the areas important for the discipline of software engineering.

The topic of the engineering in software engineering is the software architecture, the construction of software according to the software architecture

and the life cycle maintenance of the software structures built according to the architecture. That the engineering depend upon what software is being constructed, is similar to the building engineering where the building of office building and domestic buildings requires different specializations of the engineering skills. But the basic architecture and engineering training is still the same.

2.3 Software Architecture

The study of software architecture is in large part a study of software structure that began in 1968, the same year as the term Software Engineering was introduced when Dijkstra presented the work with the THE-multiprogramming system [46]. Dijkstra presented a layered software structure that supported the testability quality of the system, thereby connecting the software development test process to software architecture structures.

Twenty years later, Shaw described different styles [47]. She writes:

> *. . . important decisions are concerned with the kinds of modules and subsystems to use and the way these modules and subsystems are organized. This level of organization, the software architecture level, requires new kinds of abstractions that capture essential properties of major subsystems and the ways they interact.*

The software architecture styles Shaw describes are common ways of solving specific problems or the invention to solve one specific problem, e.g. the "Blackboard" architecture style as the solution to the speech understanding problem [48].

In the book "Software Architecture: Perspectives on an Emerging Discipline", published 1996, Shaw and Garlan describe software architecture concepts such as: components, connectors, and styles [12].

One of the frequent used definitions of Software Architecture is the definition from the book "Software Architecture in Practice" published 2003 (1st edition published 1997) written by Bass, Clements, and Kazman [13]. They define software architecture as:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

According to Gacek, Abd-Allah, Clark, and Boehm [49], a software system architecture comprises:

- A collection of software and system components, connections, and constraints.
- A collection of system stakeholders' need statements.
- A rationale which demonstrates that the components, connections, and constraints define a system that if implemented, would satisfy the collection of system stakeholders' need statements.

Gacek et al. implicitly connect the definition of sustainable development to software architecture with the third item in their list. If their system stakeholders' need statements would include the needs of future stakeholders and if the needs would include economical, social and environmental needs then their definition of software architecture would actually be in line with what is required by a software architecture for a system with sustainable development.

The standard "IEEE 1471: ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems" is the first formal standard in the area of system architecture, and was adopted in 2007 by ISO as ISO/IEC 42010:2007 [50]. In ISO/IEC 42010:2007 every system is considered in the context of its environment: the total sum of all influences determining the setting and circumstances of developmental, technological, business, operational, organizational, political, regulatory, social and any other influences upon a system. The ISO/IEC 42010:2007 definition of system architecture is:

> *The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*

This is a definition not so much in line with sustainable development as is the definition of Gacek et al [49]. "Principles" is a very vague term but could be interpreted to be principles in line with meeting current stakeholders' needs without compromising the organization's ability to meet future stakeholders' needs.

Johnson has in his PhD thesis [14], published 2002, investigated the definitions of software architecture to find a general consensus among the definitions but resorts to conclude that:

> *It is not generally agreed upon what a component or entity is, it is not generally agreed upon what a structure is, or even if it is to be called structure, and it is not generally agreed upon what else comprises software architecture.*

Considering Johnson's conclusion, the question is, how the differences in agreement affect a not risk-willing industry's adaptation of software architecture's concepts. When each industry or application area has to define its own understanding of the meaning of software architecture, it might lead to traditionally software-intensive domains taking a lead in the adaptation of software architecture concepts and the not traditionally software-intensive domains having a long way to go to reach the same software quality maturity. If software quality maturity affects the sustainability of the software system, this is a serious issue without an obvious solution. Each software application domain can hardly define its own software engineering research discipline as Jackson discusses [43].

Gacek et al. [49] and the ISO 42010 standard [50] have extended the concept of software architecture from components and their interactions to include the software architecture environment in terms of stakeholders' needs and influences like: developmental, technological, business, operational, organizational, political, regulatory, social and any other influences upon a system. These influences requires an enterprise perspective to be described and related to the software components and their interactions.

2.4 Software Architecture with an Enterprise Perspective

Enterprise[2] architecture, defined by the Federal Architecture Working Group (FAWG) [51], is: a strategic information asset base and describes the mission (i.e. the business), the information and the technologies necessary to perform the mission, and the transitional processes for implementing new technologies in response to changing mission needs. An enterprise includes interdependent resources (people, organizations, and technology) who must coordinate their functions and share information in support of a common mission. Architecture

[2]Enterprise - an organization supporting a defined business scope and mission

includes a baseline architecture[3], target architecture[4], and a sequencing plan[5].

According to Martin [52], enterprise architecture deals with "Getting to the Future" and has drivers and outcomes. The enterprise architecture is according to Martin a means for transforming enterprise objectives into business plans and mission needs.

In the mid 1990s the Department of Defense (DOD) determined that a common approach was needed for describing its architectures, so that DOD systems could efficiently communicate and inter-operate during joint and multinational operations, resulting in the DOD Architecture Framework (DODAF) [53]. The interoperability aspects of the DODAF is reflected in its architectural views which are focused on describing what's being communicated and how in the Operational View (OV) of the DODAF. The Systems View (SV) of DODAF identifies the systems that support the OVs and the Technical View (TV) describes the criteria for each required system that will satisfy the interoperability requirements. DODAF is as such not an architecture development method or a classification framework, it's an architecture description development framework focused on describing interoperability aspects of systems of systems.

TOGAF[6] is developed and maintained by members of The Open Group, working within the Architecture Forum. The original development of TOGAF Version 1 in 1995 was based on the Technical Architecture Framework for Information Management (TAFIM), developed by the US Department of Defense (DOD) [54]. The DOD gave The Open Group explicit permission and encouragement to create TOGAF by building on the TAFIM, which itself was the result of many years of development effort and many millions of dollars of US Government investment.

TOGAF is more ambitious in scope than its defense counterpart, DODAF. TOGAF organizes architectures into four domain levels: Business architecture - defines business strategy, governance, organization, and key business processes; Application architecture - specifies individual application systems to be deployed; Data architecture - defines structure of an organization's logical and physical data assets and associated data management resources; and Technology architecture - specifies software infrastructure intended to support the deployment of core, mission-critical applications.

[3] Baseline architecture - the architecture as it is today, also called as-is architecture

[4] Target architecture - the (planned) future architecture, also called to-be architecture or goal architecture

[5] Sequencing plan - the strategy for changing the baseline architecture to the target architecture, also called the transition plan

[6] http://www.opengroup.org/architecture/togaf9-doc/arch/ [accessed 12. August 2009]

Enterprise architecture descriptions have been widely adopted by the DOD but the discipline of enterprise architecture is commonly considered to have its birth in an academic article by Zachman published 1987 by the research oriented IBM Systems Journal [15]. Zachman saw the growing complexity of information software system that extended in scope and complexity to cover an entire enterprise. He stated that decentralization of system resources without structure results in chaos and argued for the need of information system architecture. Zachman searched for an objective independent base upon which to build a framework for information system architecture and resolved to be inspired by classical architecture.

In a joint article, published 1992, Sowa and Zachman explain that the Zachman framework links the concrete things in the world (entities, processes, locations, people, times and purposes) to the abstract bits in the computer [18] . The Zachman framework is not a replacement of programming tools, techniques, or methodologies but instead, it provides a way of viewing the system from many different perspectives and how they are all related. The framework logic can be used for describing virtually anything considering its history of development. The logic was initially perceived by observing the design and construction of buildings. Later it was validated by observing the engineering and manufacture of airplanes. Subsequently, it was applied to enterprises during which the initial material on the framework was published [15][16][17]. Sowa and Zachman write:

> *Most programming tools and techniques focus on one aspect or a few related aspects of a system. The details of the aspect they select are shown in utmost clarity, but other details may be obscured or forgotten. By concentrating on one aspect, each technique loses sight of the overall information system and how it relates to the enterprise and its surrounding environment. The purpose of the Information System Architecture framework is to show how everything fits together. It is a taxonomy with 30 boxes or cells organized into six columns and five rows. Instead of replacing other techniques, it shows how they fit in the overall scheme.*

According to Zachman, “Architecture” is the set of descriptive representations relevant for describing a complex object, such that the instance of the object can be created and such that the descriptive representations serve as the baseline for changing an object instance.

The columns of the framework represent different abstractions from or different ways to describe information of the complex object. The reason for

Abstraction / Perspective	INVENTORY SETS (WHAT)	PROCESS TRANSFORMATIONS (HOW)	NETWORK NODES (WHERE)	ORGANIZATION GROUPS (WHO)	TIMING PERIODS (WHEN)	MOTIVATION REASONS (WHY)
SCOPE CONTEXTS (Strategists)	e.g. Inventory Types	e.g. Process Types	e.g. Network Types	e.g. Organization Types	e.g. Timing Types	e.g. Motivation Types
BUSINESS CONCEPTS (Executive Leaders)	e.g. Business Entities & Relationships	e.g. Business Transform & Input	e.g. Business Locations & Connections	e.g. Business Role & Work	e.g. Business Cycle & Moment	e.g. Business End & Means
SYSTEM LOGIC (Architects)	e.g. System Entities & Relationships	e.g. System Transform & Input	e.g. System Locations & Connections	e.g. System Role & Work	e.g. System Cycle & Moment	e.g. System End & Means
TECHNOLOGY PHYSICS (Engineers)	e.g. Technology Entities & Relationships	e.g. Technology Transform & Input	e.g. Technology Locations & Connections	e.g. Technology Role & Work	e.g. Technology Cycle & Moment	e.g. Technology End & Means
COMPONENT ASSEMBLIES (Technicians)	e.g. Component Entities & Relationships	e.g. Component Transform & Input	e.g. Component Locations & Connections	e.g. Component Role & Work	e.g. Component Cycle & Moment	e.g. Component End & Means

Figure 4: The Zachman Framework

isolating one variable (abstraction) while suppressing all others is to contain the complexity of the design problem. Abstractions classifying the description focus are:

Inventory Sets - Describes “what” information is used

Process Transformations - Describes “How” the information is used

Network Nodes - Describes “Where” the information is used

Organization Groups - Describes “Who” is using the information

Timing Periods - Describes “When” the information is used

Motivation Reasons - Describes “Why” the information is used

The rows of the framework represent “Perspectives” classifying the description usage:

Scope Contexts - perspective descriptions corresponds to an executive summary for a planner or investor who wants an estimate of the scope of the system, what it would cost, and how it would perform.

Business Concepts - perspective is the perspective of the owner, who will have to live with the constructed object (system) in the daily routines of business. This perspective descriptions correspond to the enterprise (business) model, which constitutes the design of the business and shows the business entities and processes and how they interact.

System Logic - perspective descriptions is the designer's perspective descriptions. These correspond to the system model designed by a systems analyst who must determine the data elements and functions that represent business entities and processes.

Technology Physics - perspective descriptions correspond to the technology model, which must adapt the system model to the details of the programming languages, I/O devices, or other technology. This is the perspective where the four views of the "4+1" model by Kruchten [55] can be used to describe software architecture.

Component Assemblies - perspective descriptions correspond to the detailed specifications that are given to programmers who code individual modules without being concerned with the overall context or structure of the system.

The relevant descriptive representations would necessarily have to include all the intersections between the Abstractions and the Perspectives (Figure. 4). "Architecture" would be the total set of descriptive representations (models) relevant for describing the complex object and required to serve as a baseline for changing the complex object once it is described. Zachman's complex object is the enterprise, but principally he states that the complex object can be any object.

The Zachman framework is a structure, not a methodology for creating the implementation of the object. The Zachman Framework does not imply anything about how architecture is done (top-down, bottom-up, etc). The level of detail is a function of a cell not a function of a column. The level of detail needed to describe the Technology Physics perspective may be naturally high but it does not imply that the level of detail of the Scope Contexts descriptions should be lower or the opposite.

The framework is normalized, that is adding another row or column to the framework would introduce redundancies or discontinuities. Composite models and process composites are needed for implementation. A composite model is a model that is comprised of elements from more than one framework model. For architected implementations, composite models must be created from primitive models and diagonal composites from horizontally and vertically integrated primitives. The structural reason for excluding diagonal relationships is that the cellular relationships are transitive. Changing a model may impact the model above and below in the same column and any model in the same row.

The rules of the framework are [16]:

Rule 1: Do not add rows or columns to the framework

Rule 2: Each column has a simple generic model

Rule 3: Each cell model specializes its column's generic model

Rule 3 Corollary: Level of detail is a function of a cell, not a column

Rule 4: No meta concept can be classified into more than one cell

Rule 5: Do not create diagonal relationships between cells

Rule 6: Do not change the names of the rows or columns

Rule 7: The logic is generic, recursive

For manufacturing, a process composite would be necessary. The process composite describes the working process of creating the model descriptions of the composite model, typically ending with the descriptions of the components in the Component Assemblies perspective, e.g. a service or framework.

A third dimension of the framework, called science, has been proposed by O'Rourke et al. [56]. This extension is known as the Zachman DNA (Depth iNtegrating Architecture). In addition to the perspectives and aspects the z-axis is used for classifying the practices and activities used for producing all the cell representations.

An example of an information system classifying information standards in the Zachman framework is the Zachman ISA Framework for Healthcare Informatics Standard [57], see Figure 5.

The model, i.e. the view, in the Zachman framework can be aligned with the ISO/IEC 42010:2007 viewpoints [50]:

> *An organization desiring to produce an architecture framework for a particular domain can do so by specifying a set of viewpoints and making the selection of those viewpoints normative for any Architectural Description claiming conformance to the domain-specific architectural framework. It is hoped that existing architectural frameworks, such as the ISO Reference Model for Open Distributed Processing (RM-ODP) [58], the Enterprise Architecture Framework of Zachman [15], and the approach of Bass, Clements, and Kazman [13] can be aligned with the standard in this manner.*

Zachman's framework does not describe what language to use for the model descriptions or how to do the actual modeling for each cell. Therefore each view of the Zachman's framework is free to use the viewpoint selected by the responsible of the description. It should therefore be possible to use the viewpoints from the ISO/IEC 42010:2007 to describe a model, i.e. a view, within the framework.

The Business Concepts perspective of Zachman's framework is perhaps the most interesting to investigate for a possible integration with system architecture descriptions related to sustainable development of industrial software systems. Morris, Schindehutte, and Allen have researched the business model concept regarding the definition, nature, structure, and evolution of business models [59]. According to the authors:

> *A business model is a concise representation of how an interrelated set of decision variables in the areas of venture strategy, architecture, and economics are addressed to create sustainable competitive advantage in defined markets.*

The physical, tangible and intangible capital ensuring economical sustainability can be described within the Zachman framework. How the enterprise ensures that it minimizes the natural resource consumption can be described in a life-cycle management process described in the Process Transformations column in the Zachman framework. Subsystems' interactions can be described in the System Logic perspective in the Zachman framework and related to the business processes they support, described in the Business Concept perspective. Measures of each business process' energy consumption can therefore be related to the system's software features supporting the business process. Social capital, as networks of employees communicating with and trusting each other, may be more difficult to capture. Enterprise vision, mission and principles stating a risk-willing, open, and communicative culture can be described in

Abstraction → / Perspective ↓	INVENTORY SETS (WHAT)	PROCESS TRANSFORM. (HOW)	NETWORK NODES (WHERE)	ORGANIZATION GROUPS (WHO)	TIMING PERIODS (WHEN)	MOTIVATION REASONS (WHY)
SCOPE CONTEXTS	Description of important health service and care delivery information.	Important health care and care delivery services	Identification and description of organization and individual locations	Essential health service organizations and their functions	Identification of significant health care and care delivery events	Personal and public health impact, and care delivery business case
BUSINESS CONCEPTS	Semantic description of health care processes	Conceptual activity model of health care delivery	Structure and interrelationship of health care facilities	Healthcare information system workflow	Sequence and timelines of health care services	Personal health benefit and care delivery business objectives
SYSTEM LOGIC	Logical data model for health care information	Application architecture with function and user views	Connectivity and distributed system architecture	Health care information system human-system interface architecture	Health care event phases and process components	System functional requirements
TECHNOLOGY PHYSICS	Physical data model for health care information	System design, language specification, and structure charts	Health system information network detailed architecture	Health care information system human-system interface description	Health care information system control structures	System operational requirements
COMPONENT ASSEMBLIES	Health care information metadata, and DBMS scripts	Code statements, control blocks, DBMS stored procedures, etc	Physical data network components, addresses and communication protocols	System security architecture and operations	Health care information system component timing descriptions	Technical requirements

Figure 5: The Zachman ISA Framework for Healthcare Informatics Standard

the Scope Context's Motivation Reason column. But much of the social capital is social, since it can not be explicitly captured and transfered as economical capital in a document.

2.5 Software Architecture Environmental Influences

Shaw and Garlan suggest a software design level model consisting of *machine*, *code*, and *architecture* [12]. The machine level represents the binary software that is part of the operating system and commercial products that cannot be modified by the application developer. The code represents the program that is the domain of application development, and the third level is the architecture, which provides a model of how the system is partitioned and how the

connections between the partitions communicate.

According to Malveau and Mowray [60] this software design level model is insufficient since it does not represent any significant separation of concerns and important properties such as interoperability between systems are not considered.

Malveau and Mowray suggest a Software Design-Level Model (SDLM):

> *The Software Design-Level Model (SDLM) builds upon the fractal model. This model has two major categories of scales Å Micro-Design and Macro-Design. The Micro-Design levels include the more finely grained design issues from application (subsystem) level down to the design of objects and classes. The Macro-Design levels include system-level architecture, enterprise architecture, and global systems (denoting multiple enterprises and the Internet). The Micro-Design levels are those most familiar to developers. At Micro-Design levels, the key concerns are the provision of functionality and the optimization of performance. At the Macro-Design levels, the chief concerns lean more toward management of complexity and change. These design forces are present at finer grains, but are not nearly of the same importance as they are at the Macro-Design levels.*

Micro-Design level descriptions typically describe software components and connectors, e.g. the "4+1" view by Kruchten [55] or the four views by Soni, Nord and Hofmeister [61]. The Macro-Design level includes system-level architecture, enterprise architecture, and global systems described by e.g. the DODAF [53], TOGAF [54] or the Zachman Framework [15][16][17].

Sustainable development is about both the Micro-Design level concerns and the Macro-Design level concerns as management of complexity and change. For instance, organizational issues as an out-sourcing of development affect economical capital and social capital of the sustainable development.

The importance of technical, business, and social influences on software architecture is also discussed by Bass et al. [13]. The relationship among the technical, business, and social environments that subsequently influence future architecture is called the architecture business cycle (ABC). The ABC focuses on the creation of software architecture and the maintenance of the architecture and conformance of the system to the architecture.

An attempt to address sustainable development concerns can be found in the work of Kazman et al. [62] where the integration of established engineering methods with a development organization's life cycle is discussed. Here

the Attribute Driven Design (ADD) method by Wojcik et al. [63], and the Cost Benefit Analyze Method (CBAM)by Kazman and Ozkaya [40][62], are suggested as means for the architect to design and chose appropriate architectural responses to the new challenges during the software development life cycle. The methods are preferably used in the development phase and the Architecture Trade-off Analysis Method (ATAM), described by Clements et al [7], used after the system is released and the stakeholders want to discover risks and sensitivity points in the architecture related to business goals. The ATAM can be used to discover risks related to environmental sustainability goals if they can be expressed as quality attribute concerns, e.g. if the energy consumption of the software due to a specific architectural design can be formulated as a quality attribute concern. Usually ATAM's are concerned with economical sustainability concerns, e.g. how to make the architecture support modifiability, performance or usability in order to maintain or expand the target market with a reasonable effort [8]. The development organization's maintainability concern often implies long-term efforts, in order to improve economical sustainability, e.g. the introduction of reusable components [64] or product line system architectures [65].

The non-balance of short-term gains and long-term gains when setting business goals has been described by Dyllick and Hockerts in their article on Corporate Sustainability [4] as:

> *In recent years, driven by the stock market, firms have tended to overemphasize short-term gains by concentrating more on quarterly results than the foundation for long-term success. Such an obsession with short-term profits is contrary to the spirit of sustainability, which requires a balance between long-term and short-term needs, so as to ensure the ability of the firm to meet the needs of its stakeholders in the future as well as today.*

For the change requests entering the system after its release, the stakeholders have to take a decision if they are worth implementing or not. Considering Dyllick and Hockerts view of the importance of a balance between long-term gains and short-term gains, the change request should cover both aspects. In an article from Boehm [66], it is argued that software engineers should look at proposed changes to software systems as investment possibilities and calculate on the value of investing in those changes with methods similar to the methods in the investment economics, e.g. option theory. Especially the value of the success-critical stakeholders concerns should be considered important. For sustainable development this would mean that the software engineers must

identify the success-critical stakeholders and calculate the sustainable development related value of addressing their concerns. Ozkaya et al. propose a similar approach of using option theory when selecting architecture alternatives [40].

The implementation of change requests also have be supported by the development process. The process has to support unpredictable change requests as well as support their fast realization. The Scrum development process has gained a lot of supporters as it's a light-weight process with a strong connection to agile development methods as described by Schwaber [67]. Scrum considers the software development process to be a chaotic empirical process which requires close watching and control, with frequent intervention. A Scrum software project is controlled by establishing, maintaining, and monitoring key control parameters. The key control parameters are backlog, issues, risk, problems and changes. Scrum identifies the most important stakeholders and these success critical stakeholder's concerns are implemented at first. This is similar to Ruhe and Saliu [68] who describe the release planning approach based on the features' internal dependencies, the resource constraints and the stakeholders' importance. Scrum ignores future stakeholders' need if they are not expressed by the current stakeholders involved in the Scum process steps. In order for Scrum to be a good alternative for a company aiming at sustainable development, there must be a representative for future stakeholders' needs that has as much influence-weight as the current stakeholders. This might be difficult since there is no immediate pay-off for considering the future in terms of environmental and social capital. An additional difficulty for the Scum process to incorporate future stakeholders' needs is that it would be difficult to iterate today's system's development around changes in future stakeholders' concerns.

Curtis, Krasner, and Iscoe employed field research methods characteristic of sociology and anthropology in a field study called "A field study of the software design process for large systems" [69][70]. They studied both successful projects as well as failed projects for businesses such as: computer manufacturing, telecommunications, consumer electronics, and aerospace. Exceptional designers were shown to possess superior application knowledge and communicated well with both clients and the development team. They could translate user requirements into technology, identify unstated requirements, and mentally simulate software and interactions between parts of a system. One of the most significant findings in the study is that:

> *In particular, they (exceptional designers) envisioned how the design would generate the system behavior customers expected, even*

> *under exceptional circumstances. Yet exceptional designers often admitted that they were not good programmers, indicating they did not write optimized code, if they wrote code at all.*

The finding implies that exceptional software architects should have a solid understanding of architectural issues and customers' business processes, the application domain. The exceptional software architects increase the social capital of the company in terms of their implicit application domain knowledge.

Curtis et al. write:

> *For instance, software design is often described as a problem-solving activity. Nevertheless, few software development models include process components identified in empirical research on design problem-solving. Even worse, software tools and practices conceived to aid individual activities often do not provide benefits that scale up on large projects to overcome the impact of team and organizational factors that affect the design process.*

The findings related to management issues in Curtis's study showed that an implicit component of the managers' job was to close the gap between the technical challenges of the system and their staff's capability for solving them. Problem-solving capacity and knowledge communication is typically a part of the social sustainability researched in e.g. the educational research domain [71].

The progress of problem-solving in the software development organization can only be controlled if its being measured. Adaptations in the organization can be made if the desired sustainable development state is known and if the current state of the processes involved in achieving the goal is known. Considering the complexity in the software architecture environment, reaching the goal of sustainable development requires careful elicitation of measures to use in software development process improvement.

2.6 Software Development Measures

Software development requires a measurement mechanism for feedback and evaluation according to Basili et al. [41]. They suggest that metrics and models in industrial environments to be efficient must be focused on specific goals. The metrics must be interpreted based on an understanding of the organizational context, environment and on the specified goals. Basically, their approach is to: find the stakeholder concerns; understand what value it would

give the organization to solve the concerns; set up goals by analyzing the purpose, issues, viewpoints and objects of the concerns. The viewpoints are the stakeholders who voiced the concerns. The issue is the problem kernel of the concern and the object is the process, resource or product to which the problem kernel is related. The purpose is what the stakeholder want to do with the issue, e.g. improve, reduce, strengthen etc. Once the goal is established, questions related to the object's properties can be asked in order to find the metrics of the object.

The Goal-Question-Metric (GQM) approach could be used for sustainable development concerns voiced by stakeholders in the industrial software system's architecture environment. Goals could be set up to reach improvements on objects extracted out of the concerns.

Jain and Boehm focus on value-based software engineering (VBSE) [42]. The theory address considerations involved in the: managerial aspects of software engineering; personal, cultural, and economic values involved in developing and evolving successful software-intensive systems. Value Based Software Engineering uses success-critical-stakeholder values to situate and guide technical and managerial decisions. Similar to the GQM method, the first task is to find the stakeholders' concerns. Where Basili et al. suggest a structuring of the aspects of the concerns in order to establish goals, Jain and Boehm suggest an understanding of how the stakeholders want to win and if this way to win makes the other stakeholders to winners or losers. The negotiation between stakeholders starts with the stakeholders identifying their value propositions. The VBSE can possibly be used in conjunction with the GQM for identifying those concerns with the most sustainable development value to the organization.

Key Performance Indicators (KPIs) are quantifiable measurements, agreed to beforehand, that reflect the critical success factors of an organization. They will differ depending on the organization. Key Performance Indicators for software development could be based on the categories of identified information needs in the development organization suggested by Antolic [72]: Schedule and Progress; Resources and Cost; Product Size and Stability; Product Quality; Process Performance; Technology Effectiveness; Customer Satisfaction. Key Performance Indicators could also be based on the architectural complexity measures discussed by: Boehm et al. [73], Halstead [74] or McCabe [75].

According to Burlton [76], the type of the stakeholder, getting a value out of the process, should decide what measurement indicator is used. This is similar to the reasoning of Basili et al. [41]. For example, constructing a software architecture has a value to the architect role. Architectural complexity

could then be used as a Key Performance Indicator, to aid the architect in not constructing architectures too complex for its environment.

The view of the software architecture as a control instance working correctly only if the organizational parameters are set correctly led Dikel et al. [77] to reflect on the law developed by Ashby [78], the *law of requisite variety*, which suggests that a system should be as complex as its environment:

> *. . . in active regulation only variety can destroy variety. It leads to the somewhat counterintuitive observation that the regulator must have a sufficiently large variety of actions in order to ensure a sufficiently small variety of outcomes in the essential variables E. This principle has important implications for practical situations: since the variety of perturbations a system can potentially be confronted with is unlimited, we should always try maximize its internal variety (or diversity), so as to be optimally prepared for any foreseeable or unforeseeable contingency.*
>
> *[79]*

If a software architecture becomes more complex than its environment, it may become too expensive for the organization to support. If the environment would include the organizational environment as well as the business environment then a business domain model with a measure of the business domain complexity would be required in order to understand on what level the software architecture complexity should be. The business domain model can be described by domain analysis according to Coplien [80].

Burlton describes the maturity model of business processes in five levels [76]. Between level two and level three, architectures and KPIs should be designed. At level four, the performance of the processes are measured with the designed KPIs. At level five, the processes are continuously improved. Measuring process improvement for an industrial process, as paper production, is typically done by measuring the increase in production per time unit or in terms of observable qualities, e.g. percentage of cotton in the paper. Measuring process improvement in software development relies on KPIs measuring software development process production and/or quality being established.

Taylor Fitz-Gibbon and Lyons Morris reason around a theory-based evaluation [71]. According to the reasoning, those variables which explain the most variance in the outcomes of interest should be chosen. Transposed to software development, this would indicate that the stakeholders' concerns with the most impact on sustainable development improvement should be chosen as the basis

for establishing metrics. The theory of sustainable development would be used to indicate crucial variables in the concerns. If for e.g. the software quality attribute maintainability is a crucial variable for sustainable development of industrial software systems, then stakeholders' concerns with a maintainability object could be the basis for establishing metrics, e.g. using the Goal Question Metric approach of Basili et. al [41].

2.7 Software Architecture Quality Attributes

In [81], Barbacci et al. discuss software quality attributes. Bass et al. have introduced software quality scenarios as a way to describe system-environment interaction scenarios related to a specific quality-attribute [13] .

Maintainability is one software quality attribute important for the economical sustainability researched by e.g. Rombach [82], Oman and Hagemeister [83]. Rombach discusses maintainability at the code language level. Oman and Hagemeister have constructed a maturity attribute tree with the attributes: age, size, reuse, maintenance intensity etc.

Energy dissipation has joined throughput, area, and accuracy/precision as an important quality of the system according to Vijaykrishnan et al. [84]. Vijaykrishnan et al. argue that designers must be concerned with both optimizing and estimating the energy consumption of circuits, architectures, and software. Environmental sustainability can with this reasoning be reinforced by software architectures designed for low energy consumption.

Bass [13] points out that the software architecture quality attributes fall within two broad dimensions: those discerned by observing the system at runtime and those not observed by observing the system at runtime [13]. The former, including attributes as performance and usability, are directly influenced by the customer's concerns. The latter, such as development maintainability concerns and testability, are influenced by the development organization's concerns.

Often the quality concerns trade-off with each other. The most usable system would have no security. Security is about restricting access to system functionality and usability is about giving easy access to system functionality. The prioritization of quality concerns is depending on what business goal they support. The problem is that it's not always obvious to the system stakeholders what is the impact of their business goals concerns on the system qualities. An analysis of 24 Architecture-Tradoff-Analysis-Method (ATAM) [7] workshops and their participating stakeholders' quality attribute input is described

by Ozkaya, Bass, and Nord [8]. The ATAM uses the "Utility Tree" to describe the stakeholders' quality attribute concerns in the form of quality attribute scenarios. Ozkaya, Bass, and Nord discovered that many of the stakeholders' top 20 quality attributes, i.e. concerns, do not appear in the same fashion in common quality attribute taxonomies, e.g. the ISO 9126 [85]. Software quality attributes suffer of the same problem as software architecture: there is no common accepted semantics. This makes it harder for industrial software systems to adopt to the practice of eliciting and representing quality attribute information.

Rozanski and Woods introduce the concept of an "Architectural Perspective" as a way to modify and enhance existing views to ensure that architecture exhibits the desired quality properties [86]. Their definition of an architectural perspective is refined in the book [87]:

> *An architectural perspective is a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views.*

The security perspective activities are e.g. identify sensitive resources, define the security policies, identify threats to the system etc. Usability perspective activities are according to Rozanski and Woods: user interface design, participatory design, interface evaluation, and prototyping. To address the usability concern, Rozanksi and Woods only suggest separating the implementation of the user interface from the functional processing in contrast to the security perspective for which ten architectural tactics are discussed thoroughly.

Rozanski and Woods share the common way of describing usability for a software system as something being achieved by isolating the user interface logic from the rest of the system. Studies of software engineering projects [88][89] show that a large number of usability related change requests are made after its deployment. If usability actually requires more architectural support than user interface separation from the rest of the system logic, then the system development organization is in for a late and costly architectural redesign when these change requests hit the system.

2.8 Software Architecture's Interplay with Usability

Work in usability comes primarily from the field of Human-Computer Interaction (HCI). One bridge between the HCI field and software engineering was proposed by Jacobson, in 1987, in the form of the use case [90]. Use cases have been widely used as descriptions of how the system's user roles interact with the system. Jacobson describes the use case as: "A use case is a special sequence of transactions, performed by a user and a system in a dialog".

Constantine and Lockwood [91] write that conventional use cases typically contain too many built-in assumptions about the form of user interface that is yet to be designed. Instead they suggest the usage of a "essential use case":

> *An essential use case is a structured narrative, expressed in the language of the application domain and of users, comprising a simplified, generalized, abstract, technology-free and implementation-independent description of one task or interaction that is meaningful, and well defined from the point of view of users in some role or roles in relation to a system and that embodies the purpose or intentions underlying the interaction.*

The essential use case uses "user intentions" and "system responsibilities" instead of "user action model" and "system response model" as described by Jacobson [90][92] and Wirfs-Brock [93]. By shifting focus from actions and system responses, the essential use case abstracts the use case one more level and make it technology independent.

Task analysis and task hierarchies are often used in usability engineering. Breedvelt-Shoutern et al. have demonstrated that segments of task hierarchies can be reused [94]. Mahemoff and Johnston have investigated the topic of generic tasks [95]. Combining reuse of artifacts related to detailed software design and task models led them to the extraction of twenty-two generic tasks from the requirements for fourteen industry-based student projects. They focused on the tasks which emerged after requirements-gathering, rather than the ways in which the software supported the tasks.

Bass, John and Golden have described how practical experiences from systems with usability problems have shown that e.g. the "Cancel" function is highly important for the usability of some systems and highly difficult to implement in a released system [27][26][96]. Their research has led to the development of Usability Supporting Architecture Patterns, each addressing a

usability concern that is not addressed by separating the system's user interface from the rest of the system's functionality. In their work, John and Bass identifies a set of system-environment interaction scenarios with requirement on usability support in the architecture. The architects can use the USAPs in the early design phase to guide them in designing usable software systems.

Juristo et al. suggest an approach of using usability patterns which identify specific mechanisms that might be incorporated into a software architecture to improve the usability of the final system [31]:

> *These mechanisms have been called usability patterns and they address some need specified by a usability property. Note that usability patterns do not provide any specific software solution to be incorporated into a software architecture, they just suggest some abstract mechanism that might be used to improve usability (for example, undos, alerts, command aggregations, wizards, etc.).*

Juristo et al. use the term pattern in the sense used in the article by Perzel and Kane [97]. Perzel and Kane use the same formal description of a pattern as the software engineering domain including: problem, context, forces, classification, solution, rationale, resulting context, example, and related patterns [22]. The difference is that the Perzel's solution [97] is described as interactions between users and system, not as components and their relationship and behavior as for patterns in the software engineering domain [22].

Folmer and Bosch present an architecture-level usability assessment technique [98]. They present a scenario based assessment technique. Folmer's usability framework [98] consists of usability patterns in the sense of [97], usability properties and usability attributes. Usability properties are e.g. "minimize cognitive load" and "guidance". Usability attributes are e.g. "efficiency" and "satisfaction". Software architecture is analyzed for its support of certain usability patterns. This approach gives no support for architects wanting to create usability-supporting architectures.

John's, Bass', Juristo's, Perzel's, and Kane's formal description of a pattern is the same formal description as the software engineering domain has adapted. This way of describing patterns originates in the work of Alexander [24].

2.9 Architecture Patterns

Christopher Alexander is a building architect researcher. Alexander describes building architecture patterns as sets of forces in the world and the relations

among them [99]. In the book "The Timeless Way of Building" [24], published 1979, Alexander describes common, sometimes even universal patterns of space, events, and human existence ranging across all levels of granularity. The book "A Pattern Language" [23] contains 253 pattern entries. Each entry might be seen as an in-the-small handbook on a common, concrete architectural domain. Each entry links a set of forces, a configuration or family of artifacts, and a process for constructing a particular realization.

According to Alexander:

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*

Alexander is concerned with the life of the designed product, which is part of the usage process. The user, or customer, should, according to Alexander, experience that the thing has a rich and whole life.

The cost of the construction must be in harmony with the perceived benefit of having "life" and "feeling" and Alexander discusses this as the compression of patterns:

> *Every building, every room, every garden is better, when all the patterns which it needs are compressed as far as it is possible for them to be. The building will be cheaper, and the meanings in it will be deeper . . . It is essential then, once you have learned to use the language, that you pay attention to the possibility of compressing the many patterns which you have put together, in the smallest possible space.*
>
> (xliii-xliv, [23])

Contrasting to a belief in one optimal building process, Alexander advocates that the structure preserving of the whole of the building should dictate how the building process evolves naturally. The building process is important to Alexander as he refers to D'Arcy Thompson who insisted on that every form is basically the end result of a certain growth process [100]. The growth process must be a structure preserving transformation according to Alexander and continues to argue that the centers must be unfolded in a certain sequence: "The generative sequence is the ordering of an unfolding. It is a series of statements that describe the thing to be created". Alexander exemplifies the sequence with the sequence of the creation of a Japanese teahouse:

> *. . . if I try to locate the waiting bench too early, at a moment when I do not yet have the location of the middle barrier, the context for placing it does not yet exist. But more important, it is also not possible, in this case, for me to use the waiting bench and its location to preserve the structure of the rest. For the waiting bench to preserve the structure of the garden, I have to put it in at a time when the garden has developed. I can make the structure-preserving process work only if things come at the right time, in the right order.*

Alexander's advocated building process resembles the agile way with frequent iterations and prototyping. Many of the authors behind the "Agile Manifesto" [101] have in fact been inspired by Alexander's work on patterns: Beck and Cunningham [19]; Sutherland and Schwaber [102]. Coplien describes the emerging of the Agile discipline [103] and includes Alexander's ideas as one of the origins of the Agile discipline.

Beck et al. have described the evolution of software design patterns [104]. They write:

> *Design patterns had their origin in the late 1980's when Ward Cunningham and Kent Beck developed a set of patterns for developing elegant user interfaces in Smalltalk [19]. At around the same time, Jim Coplien was developing a catalog of language-specific C++ patterns called idioms [20]. Meanwhile, Erich Gamma recognized the value of explicitly recording recurring design structures while working on his doctoral dissertation on object-oriented software development [105]. These people and others met and intensified their discussions on patterns at a series of OOPSLA workshops starting in 1991 organized by Bruce Anderson [106][107] and by 1993 the first version of a catalog of patterns was in draft form (summarized in [108]) which eventually formed the basis for the first book on design patterns [21]. All of these activities were influenced by the works of Christopher Alexander.*

In [22], Buschman et al. define an architecture pattern as:

> *An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems[components], specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.*

Buschman et al. classify architectural patterns in:

1. Distributed Systems, e.g. broker
2. Interactive Systems, e.g. model-view-controller
3. Adaptable Systems, e.g. microkernel or reflective
4. Mud to Structure, e.g. layers, pipe and filter and blackboard

In [12], Shaw and Garlan classified architectural patterns, or styles, in common architectural styles:

1. Dataflow systems, e.g. batch sequential and pipes and filters
2. Call-and-return systems, e.g. hierarchical layers
3. Independent components, e.g. event systems
4. Virtual machines, e.g. eule-based systems
5. Data-centered systems (repositories), e.g. databases and blackboards

Enterprise application patterns differ from telecommunication application patterns according to Fowler [109]. Differences are to be found in software-hardware integration and multi-threading tasks. The complex data is the focus of an enterprise application. If the business domain of the enterprise application gets complicated, then the data gets complicated with a variety of sometimes arbitrary business rules to implement in the enterprise application system. But Fowler's conclusion that the choice of the system's architecture depends on the particular problems of the system is valid also for industrial software system. Fowler organizes the patterns into: Domain Logic patterns, Data source architectural patterns, object-relational behavioral patterns etc. Using the SDLM of Malvueau [60], the domain logic patterns capturing business logic would be Macro-Design level patterns while the others would be Micro-Design level patterns.

Fowler [109] starts his description of patterns by referring to the work of Alexander [23][24] as many have before him. The concept of a general but at the same time adaptable solution to recurring problems in a certain environment seems to speak to the heart of software engineers. The Usability-Supporting Architecture Pattern by John and Bass [27][26][96] is constructed in the same spirit, as a general but at the same time adaptable architectural solution to recurring usability problems embedded in a certain usage scenario.

Actually, the USAP is more in the spirit of Alexander than many other patterns, since Alexander is concerned with the life of the designed product, which is highly related to usability. The user, or customer, should, according to Alexander, experience that the thing has a rich and whole life. At the pattern homepage[7] of Alexander et al., the mission is described as:

> *We seek to help people design things, create things, to make them useful and beautiful, in whatever we are doing, so that we may all take part in the daily work of building a living earth.*

Alexander discovered, after some failures of people trying to apply the set of pattern (the pattern language), that the *"creative power lay in the generative structure of the language – and that lay in the sequence in which the steps were to be performed"*.

Contributing to the success or failure of a software development organization are also organization architecture patterns according to Coplien [110]. Coplien states that:

> *. . . architecture is less an echo of the tools and methods that create it than of the organization that built it. This parallelism is called Conway's law [111].*

In the Pasteur research project at Bell Labs [112], Coplien et al. investigated organizational structures. Coplien's organizational studies found two organizational patterns:

- Architecture Follows Organization, a restatement of Conway's Law [111].
- Organization Follows Location, no matter what the organizational chart says.

The second statement relates to Bürgi's research showing that social behavior and hierarchies are different in different locations in the world [113]. The organization pattern, describing communication between employees, follows the local cultural social behavior and hierarchies, no matter what the organizational chart says.

Coplien concludes [110]:

[7] http://www.patternlanguage.com/index.htm [Accessed: 10. June 2009]

> *Organizations have architecture. In fact, that's the important architecture of a system. The software architecture is kind of incidental. Software architecture is a second-order consideration; it's the people that are primary. It's critical that this perspective permeate our curricula and management policies more universally.*

Dikel et al. developed organizational principles in an effort to predict the success or failure of software architectures for large telecommunications systems [77]. In the reported case study [77], they realized that:

> *. . . technical factors, do not by themselves explain the success of a product-line architecture and that only in conjunction with appropriate organizational behaviors can software architecture effectively control project complexity.*

In [114], Kane et al. describe 30 organizational patterns and anti-patterns using the principles; Vision, Rhythm, Anticipation, Partnering and Simplification (VRAPS).

Chapter 3

Research Design

As an employee at ABB Corporate Research, I daily take part in projects with the purpose to aid the software development at the ABB units. To some extent, the work as an employee at ABB Corporate Research is the work of a consultant and to some extent, it is the work of a researcher. This thesis does not include the projects where I have "only" applied recognized methods to specific problems even if these projects have contributed to my understanding of the field of software engineering applied to industrial software systems. The case- and field studies that are part of this thesis are the ones contributing to added knowledge in the research field of software engineering in the domain of industrial software systems.

3.1 Case Study Design

Yin [115] introduces the case study as *"an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident"*. Exploratory case studies are used as initial investigations of some phenomena to derive new hypotheses and build theories, and confirmatory case studies are used to test existing theories. A precondition for conducting a case study is a clear research question concerned with how or why certain phenomena occur. This is used to derive a study proposition that states precisely what the study is intended to show, and to guide the selection of cases and the types of data to collect [116].

The case study method is visualized in Figure 6.

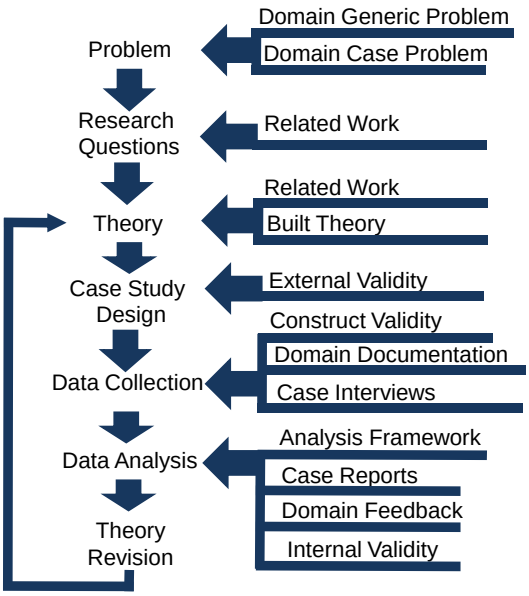

Figure 6: The research design of the System Sustainability Case Study

Four tests have been commonly used to establish the quality of any empirical social research [115]. The tests are:

Construct Validity: establishing correct operational measures for the concepts studied. For example, use multiple sources of evidence, establish chain of evidence, have key informants review draft case study report.

Internal Validity: (for explanatory or causal case studies only) establishing a causal relationship, whereby certain conditions are shown to lead to other conditions, as distinguished from spurious relationships. For example, Use logic models, do pattern matching, do explanation building, address rival explanations.

External Validity: establishing the domain to which a study's findings can be generalized. For example, use theory in single-case studies, use replication logic in multiple-case studies.

Reliability: demonstrating that the operations of a study - such as the data collection procedures - can be repeated, with the same results.

In this thesis, the Sustainable Software System study has been constructed as a case study with a multiple case design. The quality of the case study as tested by the fours tests:

Construct Validity: The case study's units of analysis were companies that: involved at least 20 developers; had software systems with a life-time of 10 years or more; and developed industrial automation applications. From May 2008 through December 2008, three automation system companies with these characteristics were visited. Three roles were interviewed at each company: senior software developer, senior software architect, and senior product manager.

Internal Validity: Not applicable since the case study is not a explanatory or causal case study.

External Validity: The domain, to which the case study findings can be generalized, is the domain of long-lived industrial software systems. The case study's three units of analysis were companies that: involved at least 20 developers; had software systems with a life-time of 10 years or more; and developed industrial automation applications. Comparison of the findings has been made with the theory proposed by Curtis et al. [70][69]. Curtis et al. conducted an extensive field study involving 19 projects in the domain of large complex software systems ranging from aerospace contractors to computer manufacturers with real-time, distributed, or embedded applications. To further strengthen the external validity the case study interview should be conducted with e.g. automotive companies also developing large complex long-lived software systems.

Reliability: Structured individual interviews were conducted which were approximately three hours long on site of the participating company. Participants were guaranteed anonymity, and the information reported has been sanitized so that no individual person or company can be identified. The same questions based on the theory in Paper **B** were asked to all of the nine interviewees. The questions were open-ended and allowed participants to formulate answers in their own terms. One person had the lead as questioner in each interview and one person had the responsibility for taking notes. After the interview, the person who had the lead

responsibility for taking notes wrote the interview protocol and sent it to the other person for review. Then the lead responsible for taking notes revised the protocol and as a last validation sent the protocol to the interviewee for review. The preliminary case study findings were presented to the participating companies and additional companies in an architecture day workshop where software architects and management were invited to discuss the findings.

3.2 Field Study Design

The Influencing Factors field study and the USAP field study were not constructed as case studies according to Yin's described case study design [115]. Instead a common research design for developing software engineering procedures or models was used, illustrated in Figure 7. The Software Engineering Taxonomy development followed the same research design as the Influencing Factors field study and the USAP field study, but the test cases were the field- and case studies of this thesis.

This thesis's field study research design is very similar to its case study research design. The case study collects data and analysis data to revise the theory, but the theory itself is not a testable method or testable solution as in this thesis's field studies. Using the theory refined in the case study, methods and solutions can be constructed but the focus of the case study is on the refinement of the theory.

In the field study research design, illustrated in Figure 7, the theory in the form of a method and an architectural pattern was tested on problem owners. The test validity was confirmed by using established usability tests and multiple test cases from the same domain. The Influencing Factors field study included two test cases to show the value of the constructed Influencing Factors method, described in Paper **A**, when the business goal prioritization or quality attribute prioritization is unclear. The test case selection included one case with unclear prioritization of business goals and one test case with unclear prioritization of software quality attributes. The goal of this field study was to show that the method makes both the business goal prioritization and the software quality attribute prioritization clear and therefore guides the architectural decisions and strengthens the stakeholders consensus around prioritized concerns. The data collection in the field study was done in form of interviews, document reading and observations from participation in project workshops and project meetings.

The USAP field study studying the interplay of usability and software ar-

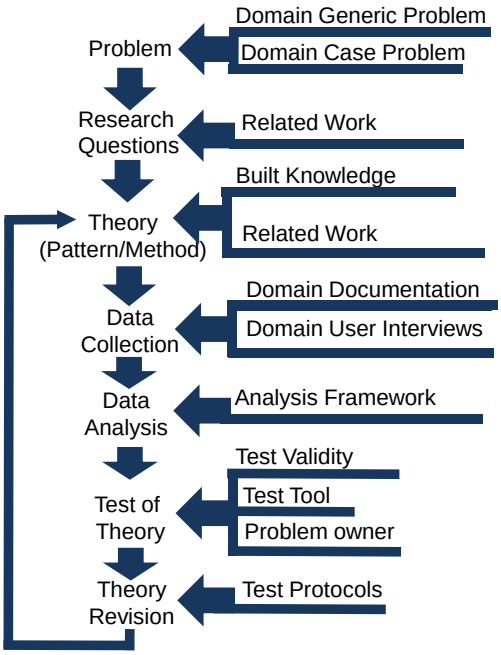

Figure 7: The research design of the Influencing Factors- and the USAP field study

chitecture [117][118][119][120] included two test cases in a sequence from different companies, but in the same industrial software system domain. The results of the first test case led to a revised USAP theory [27]. The revised USAP theory was incorporated in a USAP test tool, an experience factory. The experience factory was used in the second test case by two product-line architects and the test was documented with camera recording, queries and interviews.

One could argue that field studies, the Influencing Factors field study and the Usability-Supporting Architecture Pattern field study, fall into the category of qualitative research called action research. Action research, as described by Benbasat [121], are studies in which the author, usually a researcher, is a participant in the implementation of a system, but simultaneously wants to evaluate a certain intervention technique. This has not been the case for the

field studies in this thesis since I was not an active member of the development project teams.

The goal of the companies, participating in the field studies, has been to apply the results in their projects to get a benefit out of the participation. The problems were authentic since the problems have been contributed by the company's problem owners.

Chapter 4

Research Contribution

4.1 Influencing Factors Method

The Influencing Factors method collects concerns, extracts Influencing Factors from the concerns, and analyzes those for their influence on business goals and software quality attributes. The result is a business goal oriented prioritization of software quality attributes. The way the Influencing Factor is used in Paper **A**, the Influencing Factor is a factor that states a motivation for possible system requirements from the stakeholders' perspective.

By presenting the collected effect of several concerns, e.g. in the matrix used in Paper **A**, the Influencing Factors method makes both the business goal prioritization and the software quality attribute impact clear and therefore guides the architectural decisions and strengthens the stakeholders consensus around prioritized concerns. The analyzed concerns could also contribute to a more complete requirement specification, helping the system developers understand the origins of the requirements.

Paper **A** describes how the different impacts of the Influencing Factors are used to prioritize among the Influencing Factors for two authentic cases. The first case was performed on the upgrade of a large legacy industrial software system and the second case on the re-factoring of an existing industrial software system. The two case study systems had a diverse set of stakeholders, such as software architect, system architect, developers, testers, product management, line management, engineers, and users. Both systems suffered from an unclear understanding of what concerns were the most important. The resulting impact analysis helped the stakeholders prioritize among software quality at-

tribute scenarios in the case with the re-factored system. The prioritization included usability and led to the Usability-Supporting Architecture Pattern study described in Paper **C** and Paper **D**. The other case, with the legacy system, resulted in the stakeholders' understanding of their high focus on short-term market expansion instead of a balanced focus including long-term quality enhancements. Today this company is doing a major investment in enhancing the maintainability of the system.

The contributions of this thesis, related to research question **RQ3** "How can current and future stakeholder concerns be collected and analyzed for their impact on business goals and quality attributes in the domain of industrial software systems?", are:

- The Influencing Factors method, which shows stakeholders the impact of their concerns on the system with the intention to help stakeholders reach consensus with awareness of the impact of their concerns on business goals and quality attributes of the system.
  - The Influencing Factors method is not a design development method since it says nothing about how to translate the prioritized concerns into architectural structures.
  - The low effort required by the Influencing Factors method. For the two test cases, the gathering of concerns from stakeholders took about a person week and the contribution of each stakeholder was approximately two hours of interviews for those that couldn't participate in the one day Quality Attribute Workshop [6]. From the two case studies it was concluded that for a skilled architect with business goals understanding and software quality attributes skills, the Influencing Factors analysis of the concerns should take no longer than a day or two.

My contribution was the construction of the Influencing Factors method, the conduction of the field study investigating the Influencing Factors method's applicability to two industrial software companies, the data collection, and the analysis of the field study.

4.2 Sustainable Industrial Software Systems

The sustainable industrial software systems theory presented in Paper **B** introduces some insights into the importance of time dynamics for the sustainability of industrial software systems. The time dynamics is discussed not only

for technology factors but also for organizational and business related factors. Where the Influencing Factors method discussed business goals and their impact on architectural decisions, this paper discusses the change of business goals and their co-existence with changes in organization and market environments. This paper therefore contributes to a deeper exploration of a broader spectrum of the enterprise architecture and its relation to system- and software architecture.

The contributions of this thesis, related to research question **RQ2** "What are the concerns affecting the sustainable development of an industrial software system?", are:

- The industrial software system sustainability theory that states that:
  - The most important factor to recognize for sustainable development is the factor of change. Change in organization, technology, and market over time is something inevitable and must be managed.
  - The second most important factor, is the sustainable target market. Customers needing the same basic functionality over decades, tend to invest in systems that have sustained on the market for long times. Sustainable systems are rewarded by the sustainable market, thereby increasing their sustainability further.

The industrial software system sustainability theory is a common contribution by me and Anders Wall.

4.3 Usability-Supporting Architecture Patterns

Usability and its interplay with software architecture was discussed in the Influencing Factors paper, Paper **A**, as one of five quality attributes. Paper **C** reports on and discusses the Usability-Supporting Architecture Pattern study in the domain of sustainable industrial software systems and contributes with a description of an enhanced research method and a software tool that visualizes the research method's constructed responsibilities. The tool, visualizing the responsibilities, acts as an experience factory [122] housing reusable architectural knowledge for a set of system-environment interaction scenarios. The architects access the reusable experience in the form of reusable responsibilities with implementation instructions. The architects use the knowledge as instructions on how to implement usability support in the software architecture early in the software design phase.

In Paper **C**, it is reported on the revised USAP method and the construction of the tool that visualized the method's results. In Paper **D**, it is reported on and discussed the results and validation of the USAP case study. The contribution of this paper is significant since very few studies can report on software architects being able to use a tool early in the software design in a way that helps them implement usability support in the software architecture. The two architects used the tool for one day and reported on a development cost saving of more than five weeks for the one-day interaction with the tool, giving a return-of-investment of 25:2.

The contributions of this thesis, related to research question **RQ1** "How can support for usability be built into software architecture of industrial software system in the early design phase?", are:

- The identification of four foundational patterns describing reusable activities and tasks with architectural usability-supporting responsibility descriptions and responsibility implementation descriptions.
- Three Usability-Supporting Architecture Patterns: "Alarm & Event", "User Profile" and "Environment Configuration".
- The experience factory, in the form of a web-based tool, containing the reusable architectural knowledge. Architects access the knowledge in order to understand how to implement usability support in the architecture early in the design phase.
  - The experience factory can be used for evaluating an architecture with respect to its usability support.
  - The company, using the experience factory for their product-line-system's architecture, reported on a Return-Of-Investment of 25:2 and an improved architecture quality as a result of using the experience factory.
- Presenting a sequence of responsibilities, with responsibility implementation descriptions, to the architects in a step-by-step manner is perceived as much more relevant and usable than being presented with an sample solution in the form of a UML pattern.
  - The field study experienced success after refactoring the UML pattern embedding the reusable architectural responsibilities into a sequence of steps. Each step's responsibility description instructs the

architect how to architecturally support a part of the common tasks of the system's three USAPs.

– The architects felt this way of presenting a pattern helped them reflect on their own architectural design and take appropriate design decision in relation to each responsibility in a natural order of steps. This success is in line with Alexander's discovery that people trying to apply the set of pattern (the pattern language), achieved success first when using a generative structure of the language - the sequence in which the steps were to be performed.

My contribution was the field study's project management, the experience factory's architectural design and implementation, and the discovery of the usability supporting architectural responsibilities importance for product line system's architecture. My, Lövemark's and Alfredsson's common contribution was: the identification of the common tasks for the "Alarm & Event" process, the construction of reusable usability supporting architectural responsibilities for the tasks and the conduction of the first test case in the field study. My, John's, Bass's and Golden's common contribution was: the systems' task analysis of the processes of creating system-environment work products; the discovery of common tasks among the processes of creating four system-environment work products; and the replacement of a UML sample solution with a generative sequence of reusable architectural responsibility descriptions with responsibility implementation descriptions.

4.4 Software Engineering Taxonomy

Since sustainable development must address concerns from the Macro-Design level down to Micro-Design level, a framework that can classify the concerns would be very useful in order to find interrelationships between the concerns for the construction of strategies to improve the sustainable development. Therefore three Enterprise Architecture frameworks were considered. The three frameworks were: the Zachman framework [15][16][17] , the Department Of Defense Architecture Framework (DODAF) [53] and The Open Group Architecture Framework (TOGAF) [54].

As this thesis was searching for an enterprise architecture artifact classification framework, not an enterprise architecture description development framework focusing on interoperability aspects or in-house information system architecture development framework, it resorted to study the Zachman framework in more detail.

Paper **E** describes the motivation, the assumptions, and the creation of the Software Engineering Taxonomy. The assumptions made it possible to construct the Software Engineering Taxonomy as a derivative of the Zachman Framework. The paper also classifies all software engineering artifacts from the IEEE Software Engineering Book Of Knowledge (SWEBOK) published 2004 [123], to test the completeness of the classification capacity of the taxonomy.

Apple and Google are test cases showing how shared composite models crossing the site dimension of the Software Engineering Taxonomy might lead to faster innovation.

The Scrum process artifacts are classified to show which Software Engineering Taxonomy perspective is the focal point of the Scum process. The result is a large set of Scrum artifacts being classified in the software development organization's Business Concept perspective.

The contributions of this thesis, related to research question **RQ4** "How can industrial software system stakeholders' concerns be described by views in an enterprise framework, that incorporates software engineering artifacts descriptions?", are:

- The Software Engineering Taxonomy derived out of the Zachman Framework.
  - The Software Engineering Taxonomy integrates software engineering artifacts into the views of the Zachman framework, thereby building relations between enterprise views and software engineering views for industrial software systems.
  - The Software Engineering Taxonomy adds the site dimension to the Zachman Framework. The site identifies the environment of the system descriptions, e.g. the operational environment or the development environment. The development environment can be shared between development organizations resulting in multiple sites sharing view descriptions.
  - The classification of the IEEE SWEBOK [123] artifacts uses only one software development environment perspective, not the operational environment perspective, resulting in the classification being two-dimensional.
  - The Software Engineering taxonomy can serve as a reasoning framework into which concerns, artifacts and results of software engineering theories, processes and case studies are classified for fur-

ther analysis. The consistency rules of the Zachman framework are valid also for the Software Engineering Taxonomy.

My contribution was the construction of the Software Engineering Taxonomy, the classification of the IEEE SWEBOK software engineering artifacts in the Software Engineering Taxonomy, and the analysis of Apple, Google, and Scrum cases guided by the classification of their software engineering artifacts in the Software Engineering taxonomy.

4.5 Applied Software Engineering Taxonomy

Paper **F** uses the Software Engineering Taxonomy from Paper **E** as a reasoning framework to analyze the artifacts from: the Influencing Factors method study, the Usability-Supporting Architecture Patterns study and the System Sustainability case study. Case study design, execution and analysis of the System Sustainability case study is additionally described in Paper **F**.

The contributions of this thesis, related to research question **RQ4** "How can industrial software system stakeholders' concerns be described by views in an enterprise framework, that incorporates software engineering artifacts descriptions?" are:

- The Software Engineering taxonomy can classify all artifacts from this thesis's three studies' collected data and used theory.
- Not all of the Software Engineering Taxonomy views are necessary to describe a specific method or theory. What views are used, depends on the scope of the researched object. In the classification of the USAP study artifacts, eight views were used in contrast to the Sustainable System study that used nineteen views.
- The Software Engineering Taxonomy made the site distinction between system-operational environment and system-development environment very clear for the case/ field study artifacts. The site distinction decides what roles and work products are related to what system-environment interface. For example, in the operational environment, the system-environment interface may be the user interface and the work product an "Alarm & Event Condition". In the system-development environment the interface may be the development environment's interface, e.g. the implementation's interface to the system and the work product a code structure.

The contributions of this thesis, related to the extended analysis of the research presented in Paper **C** and Paper **D**, to the research question **RQ1** "How can support for usability be built into software architecture of industrial software system in the early design phase?" are:

- The inclusion of a traditional enterprise perspective, the business concepts perspective, led to discoveries of new interrelationships between the USAP artifacts: system-environment interaction scenario, system environment business roles & work products, system-environment activities and tasks related to the roles & work products, responsibility descriptions, quality attributes, and responsibility implementation descriptions.

- Classification of the USAP artifacts made use of the business concept perspective for four of the twelve artifacts. The inclusion of a traditional enterprise perspective, the business concepts perspective, led to new conclusions regarding the use of general activities for pattern creation.

- System environment business roles and work products are a key artifact in linking the USAP scenario [26] to common activities and tasks supporting more than one role or more than one work product.

- System environment may be operational or development environment. The environment decides what system-environment interface, business roles and work products should be used in the USAP information description/ selection process. The environment dimension, the site dimension in Paper **E**, of the Software Engineering Taxonomy is therefore important.

- The placeholder of the common activity is furnished by the work product or the role for the three USAP scenarios in the field study.

- The responsibility is related to the quality chosen to be supported for the scenario. For USAP, the usability quality is supported by the USAP responsibility.

The contributions of this thesis, related to the extended analysis of the Sustainable Industrial Software Systems from Paper **B**, to the research question **RQ2** "What are the concerns affecting the sustainable development of an industrial software system? development organization" are:

- The sustainable key-competences in the industrial software system development organization carry the application domain knowledge and the system knowledge, thereby increasing the social sustainability of the company. The sustainable key-competence pass the knowledge on to the system developers during informal design discussions.
- The development organizations sustain economical capital by planning for changes when the changes are technology changes. When the changes are organizational, e.g. distributed development, the management have lost social capital by failing to plan for how the development organization has to adapt to the new work-form. It has been too little known in the companies, what requirements a distributed development environment has on the development organization's structures and communication.
- The incorporation of a remotely located development team in the development organization will be especially difficult in a culture that has social capital invested in sustainable key-competences and their informal spreading of knowledge. If the organization has ignored investigating in explicit software documentation, increasing the tangible economical capital, the new remotely located team can make use of neither the social capital nor the economical capital related to system know-how.
- The sustainable target market increases the intangible economical capital.
- Intangible economical capital in the form of goodwill and reputation is increased by delivering reliable systems for a long-time to the target markets.
- The business case arguing added value of software engineering for sustainable development is not good enough for the three investigated cases making the use of software engineering methods and artifacts sparse.
- Curtis's study [70][69], the Dikel study [77] and the Sustainable Industrial Software Systems case study point toward a conclusion that sustainable development concerns related to the software development organization, must be addressed first before software engineering tools and methods can have a significant impact on sustainable development.

The contributions of this thesis, related to the extended analysis of the Influencing Factors from Paper **A**, to the research question **RQ3** "How can current and future stakeholder concerns be collected and analyzed for their impact

on business goals and quality attributes in the domain of industrial software systems?" are:

- The perspective of the influencing factor is connected to the quality concern's ownership of the influencing factor. If the customers own the quality concern, i.e. has voiced the quality concern, corresponding influencing factor can be found in all the Software Engineering Taxonomy perspectives. The development organization's maintainability concern's corresponding influencing factors are only found in the Business Concepts and Scope Contexts perspective, not in the System Logic perspective. Discussions regarding architectural solutions to maintainability issues, seem to be reserved for the architects and developers outside success-critical stakeholders' discussion forums.

My contribution to Paper **F** was the analysis of: the Influencing Factors method field study, The USAP field study, and the Sustainable Industrial Software Systems case study. Our common contribution was the design and data collection of the Sustainable Industrial Software Systems case study.

Chapter 5

Future Work

The most substantial future work is related to the system sustainability theory and the USAP's integration of additional quality attributes.

5.1 Sustainable Industrial Software Systems

It remains to expand the external validity of the Sustainable Industrial Software Systems case study, i.e. find more related work and to include a domain-external case with a long-lived complex software system, e.g from the automotive domain. The automotive domain would be especially interesting due to its high requirements on environmental sustainability. The theory in Paper **B** was the base for the propositions. Since the propositions were not all verified, the theory should be modified according to the findings from the case study and validated. The analyzed findings should also be further discussed with the involved case companies.

When applying the concept of sustainable development to the classified concerns from the interviews, which were ranked as being of high importance, there was an unbalance between the economical sustainability, environmental sustainability, and the social sustainability. Most of the concerns addressed economical capital or ways of increasing economical capital. Some concerns addressed social capital but no concern addressed environmental capital. In the analysis, one environmental capital concern is added based on knowledge of the systems, collected through documentation and experience. When the value of addressing the individual sustainable development concern is not known,

it's very difficult to say, if the system development is sustainable or not. The concerns with the highest impact on sustainable development must be found, based on the added value of addressing the concerns. Metrics for the objective of the concerns' issues can be established, possibly using the Goal Question Metric approach suggested by Basili et al. [41].

Another open issue is the value of software engineering concepts to the domain of industrial software systems. The lack of organization prerequisites for using software engineering concepts led to software architecture related issues being underrepresented in the Sustainable System case study interviews. But that does not mean that software engineering concepts do not have any impact on sustainable development of industrial software systems. The sequence of introducing them must simply start with raising software engineering awareness among the executive leaders, who in their turn could raise the software engineering knowledge among the staff. By doing so, the social capital of the organization would be increased as well as its social sustainability. But the current state leaves the sustainability case study with open questions around the importance of software architecture concepts for sustainable development. A case study can simply not answer these questions, since the domain must mature and use software architecture concepts on a daily basis for some time, before their importance for sustainable development can be evaluated.

Will the future stakeholders have their needs met without the systems incorporating software engineering concepts as: explicit software development process, formal software evaluation process, domain analysis, software documentation, pattern languages, model driven architecture etc? If not, future work must formulate a sustainability business case that includes the value of software engineering. To do so, a gap analysis between the current state today and the desired state must be done and measures constructed for the progress of the movement to the desired state. Software engineering would be the tool that enforces the movement to the desired state. Enterprise architecture and usability are concepts, important for sustainable development, that need to be incorporated with software engineering.

It remains to use the Software Engineering Taxonomy classification of sustainable development concerns for the set-up of goals and metrics in order to address some of the sustainable development concerns the companies felt they could meet in a better way. The interrelationships between the classified concerns could then be used to create an sustainable development improvement process, in the same manner as the USAP information description/ selection process was created in Paper **F**.

5.2 Usability Supporting Architecture Patterns

The Usability-Supporting Architecture Patterns study is currently being extended in order to apply the enhanced concepts. The new field study will use security and safety as research base. If multiple quality attribute supporting responsibilities could be created then the issue of quality attribute trade-offs will surface. Further, the question of how to identify and present conflicting quality concerns to the architects will have to be answered. The following reasoning is part of an ongoing case study were no results are yet published.

In the activity taxonomy, the quality aspect is to be found on the responsibility level as shown in Figure 7. For example, the “Alarm & Event” tasks have usability requirements as well as security- and safety requirements. Not all operators or system engineers are allowed to author an “Alarm & Event” condition since it will have implications on the safety and security of the environment of the “Alarm & Event” system. A falsely authored “Alarm & Event” condition might lead to the “Alarm & Event” system not warning the operator when the devices are not working properly, possibly causing damages in the environment. Additionally the “Alarm & Event” might not warn the operator about an intrusion attempt to the system. The responsibilities handling security and safety for the activity task “Create a specification” must be considered as well as the usability responsibilities for the same activity task.

The architects may assign the responsibility to a portion of the system, e.g. a component which they put an identifier on, e.g. a name or number. If the architects have assigned e.g. both a security and a usability responsibility to the same component(s), then the trade-off between the responsibilities can be made visible on component level.

Figure 8 shows how the activity task “Modify a specification” has two quality concerns: security and usability. The system’s specifications are only allowed to be modified by authorized users due to security issues and possibly also safety issues. To address the security concern of the activity task, one security responsibility states “The system must permit or prohibit specific authoring of a specification”. At the same time one usability responsibility states “The system must provide a way to access the specification”. These two responsibilities have different implementation descriptions. The security responsibility’s implementation description says that there should be portions of the systems that permit or prohibit access depending on who asks for permission to modify the specification. The usability responsibility’s implementation description says that there must be portions of the system that provide access to the specification.

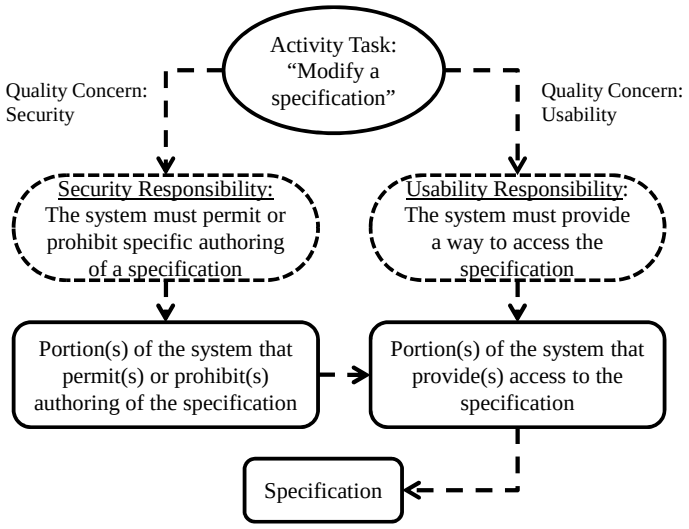

Figure 8: Example of activity task's multiple quality concerns' trade-off

In a ongoing case study, the described trade-off concept will be implemented in an extended version of the experience factory, the USAP web test tool, described in Paper **C** and Paper **D**.

Bibliography

[1] P. Pollan. Our decrepit food factories. *New York Times*, 2007.

[2] G.C Unruh. Escaping carbon lock-in. *Energy Policy*, vol. 30(no.4):pp. 317–325, 2002.

[3] G.H. Brundtland. Our common future. Report of the World Commission on Environment and Development. Published as Annex to General Assembly document A/42/427, 1987.

[4] T. Dyllick and K. Hockerts. Beyond the business case for corporate sustainabilityt. *Business Strategy and the Environment*, 11:130–141, 2002.

[5] C. Labuschagne, A.C. Brent, and R.P.G. Erck. Assessing the sustainability performances of industries. *Journal of Cleaner Production, Volume 13, Issue 4, March 2005, Pages 373-385*, 13(4):373–385, 2005.

[6] M. Barbacci, R. Ellison, A. Lattance, J. Stafford, C. WeinStock, and W. Wood. Quality attribute workshops, 3rd edition. Technical report, Software Engineering Institute, Pittsburgh, PA, USA, 2003.

[7] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures, Methods and Case Studies*. Addison-Wesley, Boston, 2002.

[8] I. Ozkaya, L. Bass, R.L. Nord, and R.S. Sangwan. Making practical use of quality attribute information. *Software, IEEE*, 25(2):25–33, March-April 2008.

[9] D.J. Reed. Stalking the elusive business case for corporate sustainability. World Resources Institute, Washington, 2001.

[10] O. Salzmann, A. Ionescu-Somers, and U. Steger. The business case for corporate sustainability:: Literature review and research options. *European Management Journal*, 23(1):27 – 36, 2005.

[11] R K. Singh, H.R. Murty, S.K. Gupta, and A.K. Dikshit. An overview of sustainability assessment methodologies. *Ecological Indicators*, 9(2):189 – 212, 2009.

[12] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[13] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, second edition, 2003.

[14] P. Johnsson. *Enterprise Software System Integration: An Architectural Perspective*. PhD thesis, Industrial Information and Control Systems, Royal Institute of Technology (KTH), Stockholm, Sweden, 2002.

[15] J. A. Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[16] J. A. Zachman. *The Zachman Framework for Enterprise Architecture; A Primer for Enterprise Engineering and Manufacturing*. Zachman International, 2003.

[17] J. A. Zachman. The Zachman Framework and Observations on Methodologies. *Business Rules Journal*, 5(11), 2004.

[18] J. F. Sowa and J. A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM System Journal*, 31:590–616, 1992.

[19] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. Technical Report Technical Report No. CR-87-43, Apple Computer, Inc. and Tektronix, Inc., 1987. Submitted to the OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming.

[20] J. O. Coplien. *Advanced C++: Programmmg Styles and Idioms*. Addison-Wesley, 1992.

[21] E. Gamma, R Helm, R. Johnson, and J. Wissides. *Design Patterns - Elements of Reusable Object-Oriented Sojlware*. Addison-Wesley, 1995.

[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture A System of Patterns*, volume 1. Wiley, first edition, 1996.

[23] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA, 1977.

[24] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[25] L.J. Hoffman, K. Lawson-Jenkins, and J. Blum. Trust beyond Security: An expanded Trust Model. *Commun. ACM*, 49(7):95–101, 2006.

[26] L. Bass, B. E. John, and J. Kates. Achieving usability through software architecture. Technical Report No. SEI-TR-2001-005, Carnegie Mellon University/Software Engineering Institute, Pittsburgh, PA, 2001.

[27] L. Bass and B. E. John. Linking usability to software architecture patterns through general scenarios. *The Journal of Systems and Software*, 66:187–197, 2003.

[28] E. Folmer and J. Bosch. Architecting for usability: a survey. *Journal of Systems and Software*, 70(1-2):61–78, 2004.

[29] E. Folmer, J. van Gurp, and J. Bosch. A Framework for capturing the Relationship between Usability and Software Architecture. Software Process: Improvement and Practice, Volume 8, Issue 2. Pages 67-87., 2003.

[30] N. Juristo, H. Windl, and L. Constantine. Introducing usability. *Software, IEEE*, 18(1):20–21, Jan/Feb 2001.

[31] N. Juristo, M. Lopez, A. Moreno, and M.-I. Sanchez-Segura. Improving software usability through architectural patterns. Paper presented at the ICSE 2003 Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction, Portland, Oregon, USA., 2003.

[32] N. Juristo, A.M. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *Software Engineering, IEEE Transactions on*, 33(11):744–758, Nov. 2007.

[33] K.E. Sveiby. *The New Organizational Wealth: Managing and Measuring Knowledge Based Assets*. Berrett Koehler, San Francisco, CA., 1997.

[34] R. Ayres. *Industrial Metabolism: Restructuring for Sustainable Development*, chapter Industrial Metabolism: Theory & Policy, pages 3–20. United Nations University Press, 1994.

[35] G. Strand. Keyword evil: Google's addiction to cheap electricity. *Harper's Magazine*, March 2008.

[36] D. Dunphy, A. Griffiths, and S. Benn. *Organizational Change for Corporate Sustainability: Understanding Organizational Change*. Routledge, 2003.

[37] J.S. Coleman. Supplement: Organizations and institutions: Sociological and economic approaches to the analysis of social structure. *The American Journal of Sociology*, 94:S95–S120, 1988.

[38] Martin O'Connor. The "four spheres" framework for sustainability. *Ecological Complexity*, 3(4):285 – 292, 2006. Complexity and Ecological Economics.

[39] B. Ness, E. Urbel-Piirsalu, S. Anderberg, and L. Olsson. Categorising tools for sustainability assessment. *Ecological Economics*, 60(3):498 – 508, 2007.

[40] I. Ozkaya, R. Kazman, and M. Klein. Quality-attribute based economic valuation of architectural patterns. In *Economics of Software and Computation, 2007. ESC '07. First International Workshop on the*, pages 5–5, May 2007.

[41] V. R. Basili, G. Caldiera, and D. H. Rombach. *Encyclopedia of Software Engineering*, chapter The goal question metric approach. Wiley, 1994.

[42] A. Jain and B. Boehm. Developing a theory of value-based software engineering. In *EDSER '05: Proceedings of the seventh international workshop on Economics-driven software engineering research*, pages 1–5, New York, NY, USA, 2005. ACM.

[43] M. Jackson. Will there ever be software engineering? *IEEE Software*, pages 36–39, 1998.

[44] H. Ziv and D.J Richardson. The Uncertainty Principle in Software Engineering. In *19th International Conference on Software Engineering (ICSE'97)*, 1997.

[45] V. R. Basili and J. D. Musa. The future engineering of software: A management perspective. *Computer*, 24(9):90–96, 1991.

[46] E. Dijkstra. The structure of the "THE"-multiprogramming system. *Commun. ACM 11*, 5:341–346, 1968.

[47] M. Shaw. Larger scale systems require higher-level abstractions. Proceedings of the Fifth International Workshop on Software Specification and Design, 1989.

[48] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and R. D. Raj. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Comput. Surv.*, 12(2):213–253, 1980. 356816.

[49] C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm. On the definition of software system architecture. In *ICSE 17 Software Architecture Workshop*, 1995.

[50] R. Hilliard. Systems and software engineering - Recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1–24, 15 2007.

[51] R. C. Thomas. A Practical Guide to Federal Enterprise Architecture,. www.gao.gov/bestpractices/bpeaguide.pdf, 2001. retrieved July 11th 2009.

[52] J. N. Martin. *An introduction to the Architectural Frameworks DODAF/MODAF/NAF*. Course given at the Royal Institute of Technology, Stockholm, Sweden, 2006.

[53] DoD. Department of Defence Architecture Framework Working Group, DoD Architecture Framework, DoDAF, version 1.0. Department of Defence, 2003.

[54] TOG. *The Open Group Architecture Framework, version 8/9, 2002/6*. The Open Group,.

[55] P. B. Kruchten. The “4+1” View Model of architecture. *Software, IEEE*, 12(6):42–50, Nov 1995.

[56] C. O’Rourke, N. Fishman, and W. Selkow. Enterprise Architecture, Using the Zachman Framework. *Thomson Course Technology*, 2003.

[57] M. Diehl. Zachamn ISA Framework for Healthcare Informatics Standard. Available: http://apps.adcom.uci.edu/EnterpriseArch/ Zachman/Resources/ ExampleHealthCareZachman.pdf [Accessed 25. September 2009], 1997.

[58] ISO/IEC 10746 - 3: 1996, Information technology - Open distributed processing - Reference model: Architecture, 1996.

[59] M. Morris, M. Schindehutte, and J. Allen. The entrepreneur’s business model: toward a unified perspective. *Journal of Business Research*, 58(6):726 – 735, 2005. Special Section: The Nonprofit Marketing Landscape.

[60] R. Malveau and T. J. Mowbray. *Software Architect Bootcamp*. Prentice Hall Professional Technical Reference, 2003.

[61] D. Soni, R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *ICSE ’95: Proceedings of the 17th international conference on Software engineering*, pages 196–207, New York, NY, USA, 1995. ACM.

[62] R. Kazman, J. Asundi, and M. Klein. Making architecture design decisions: An economic approach. Technical report, Software Engineering Institute, Carnegie Mellon University, 2002.

[63] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-driven design (add), version 2.0. Technical Report CMU/SEI-2006-TR-023 ESC-TR-2006-023, Software Engineering Institute, Pittsburgh, USA, 2006.

[64] I. Jacobson, M. Griss, and P. Jonsson. Making the reuse business work. *Computer*, 30(10):36–42, Oct 1997.

[65] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[66] B.W. Boehm and K.J. Sullivan. Software economics: a roadmap, 2000.

[67] K. Schwaber. Scrum development process. Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger 6(4), October 1995.

[68] G. Ruhe and M.O. Saliu. The art and science of software release planning. *IEEE Software*, 22:47–53, 2005.

[69] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. Communications of the ACM, Vol. 31 No. 11, pp. 1268-87., 1988.

[70] W. Curtis, H. Krasner, V. Shen, and N. Iscoe. On building software process models under the lamppost. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 96–103, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[71] C. Taylor Fitz-Gibbon and L. Lyons Morris. Theory-based evaluation. *Evaluation Practice*, 17(2):177 – 184, 1996.

[72] Z. Antolic. An Example of Using Key Performance Indicators for Software Development Process Efficiency Evaluation. Technical Report, R&D Center, Ericsson Nikola Tesla d.d., 2008.

[73] B. Boehm, Abts C., A. Winsor Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[74] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[75] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[76] R. Burlton. *Business process management: profiting from process*. Sams, Indianapolis, IN, USA, 2001.

[77] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying software product-line architecture. *Computer*, 30(8):49–55, Aug 1997.

[78] W. R. Ashby. *An Introduction to Cybernetics*. First Edition, Chapman and Hall: London, UK, 1956.

[79] W.R. Ashby. The Law of requisite Variety. Avalable: http://pespmc1.vub.ac.be/REQVAR.html [accessed 20. August 2009].

[80] J. O. Coplien. *Multi-Paradigm Dedign for C++*. Addison-Wesley, Reading, MA, 1998.

[81] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock. Quality Attributes. Technical Report CMU/SEI-95-TR-021, CMU/SEI, 1995.

[82] H.D. Rombach. A controlled expeniment on the impact of software structure on maintainability. *Software Engineering, IEEE Transactions on*, SE-13(3):344–354, March 1987.

[83] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344, Nov 1992.

[84] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 95–106, New York, NY, USA, 2000. ACM.

[85] IS0 9126-1:2001*Information Technology - Software engineering - Product quality - Part 1: Quality model*, International Organization for Standardization, 2001.

[86] E. Woods and N. Rozanski. Using architectural perspectives. In *Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005)*, pages 25–35, Pittsburgh, Pennsylvania, USA, November 2005. IEEE Computer Society.

[87] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives*. Addison-Wesley, 2005.

[88] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, NY,, 1992.

[89] T. K. Landauer. *The Trouble with Computers: Usefulness, Usability and Productivity*. MIT Press., Cambridge, 1995.

[90] I. Jacobson. Object oriented development in an industrial environment. In *OOPSLA '87: Object-Oriented Programming Systems, Languages and Applications*, volume 22(12), pages 183–191. SIGPLAN Notices, 1987.

[91] L L. Constantine and L. A D. Lockwood. *Software for User: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley, 1999.

[92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1992.

[93] R. Wirfs-Brock and A. McKean. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley, 2003.

[94] I. M. Breedvelt-Schouten, Paternò. F., and C. Severijns. Reusable structures in task models. In *DSV-IS*, pages 225–239, 1997.

[95] M. J. Mahemoff and L. J. Johnston. Brainstorming with generic tasks: An empirical investigation. In *Interfacing Reality in the New Millennium: OZCHI 2000*, pages 224–231, Sydney, December 2000. ACM.

[96] E. Golden, B. E. John, and L. Bass. The value of a usability-supporting architectural pattern in software architecture design: A controlled experiment. In *Proceedings of the 27th International Conference on Software Engineering, ICSE*, St. Louis, Missouri, May 2005.

[97] K. Perzel and D. Kane. Usability patterns for applications on the world wide web. In *PloP '99 Conference.*, 1999.

[98] E. Folmer, J. Van Gurp, and J. Bosch. Software architecture analysis of usability. In *IFIP Working Conference on Enginering for Human-Computer Interaction*, 2004.

[99] R. P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, New York Oxford, 1998.

[100] D.A.W. Thompson. *On Growth and Form*. Cambridge Univ. Press, Cambridge, 1917.

[101] K. Beck, K. Schwaber, W. Cunningham, M. Fowler, and J. et al Sutherland. *The Agile Manifesto*. Available: http://agilemanifesto.org/ [accessed 10. February 2009].

[102] J. Sutherland and K. Schwaber. The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process. Available: www.jeffsutherland.com/scrum/ScrumPapers.pdf [Accessed 25. June 2009], October 2007.

[103] J. O. Coplien. *For those that were Agile before Agile was cool*. Keynote speech at OO Days at Tampere University of Technology, November 2008.

[104] K. Beck, R. Crocker, G. Meszaros, J. Vlissides, J. O. Coplien, L. Dominick, and F. Paulisch. Industrial experience with design patterns. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 103–114, Washington, DC, USA, 1996. IEEE Computer Society.

[105] E. Gamma. *Object-Oriented Software Development based on ET++*. PhD thesis, University of Zurich, Institut fur Informatik, 1991.

[106] B. Anderson and P. Coad. Patterns workshop. In *In 00PSLA '93 Addendum to the Proceedings*. ACM Press., January 1994.

[107] B. Anderson. Towards an architecture handbook. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 167–168, New York, NY, USA, 1992. ACM.

[108] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 406–431, London, UK, 1993. Springer-Verlag.

[109] M. Fowler. *Pattern Of Enterprise Application Architecture*. Addison-Wesley, 2003.

[110] J. O. Coplien. Organization and architecture. 1999 CHOOSE Forum on Object-oriented Software Architecture, 1999.

[111] M. E. Conway. How do committees invent? *Datamation magazine*, 1968.

[112] J. O. Coplien. Borland software craftsmanship: A new look at process, quality and productivity. In *5 th Annual Borland International Conference*, 1994.

[113] P. Bürgi. Seeing Work Practices Through a Cultural Lens. *Next Practice*, 3(1), 2004.

[114] D. Kane, D. Dikel, and J. Wilson. *Software Architecture: Organizational Principles and Patterns*. Prentice Hall, 2001.

[115] R. K. Yin. *Case study research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. SAGE Publications, third edition, 2003.

[116] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In F. Shull, J. Singer, and D. I. K. Sjöberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 258–311. Springer London, 2008.

[117] P. Stoll, L. Bass, B. E. John, and E. Golden. Preparing Usability Supporting Architectural Patterns for Industrial Use. Proceedings of International Workshop on the Interplay between Usability Evaluation and Software Development (I-ISED), Pisa, Italy, 2008.

[118] P. Stoll, F. Alfredsson, and S. Lövemark. Usability Supporting Architecture Pattern for Industry. Proceedings of the NordiCHI 2008, Lund, Sweden, 2008.

[119] P. Stoll, L. Bass, B.E. John, and E. Golden. Supporting Usability in Product Line Architectures. Proceedings of the 13th International Software Product Line Conference (SPLC), San Francisco, USA, August 2009.

[120] B. E. John, L. Bass, E. Golden, and P. Stoll. A responsibility-based pattern language for usability-supporting architectural patterns. Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS), Pittsburgh, PA, US, 2009.

[121] I. Benbasat, D. K. Goldstein, and M. Mead. The case research strategy in studies of information systems. *MIS Q.*, 11(3):369–386, 1987.

[122] V.R. Basili, G. Caldeira, and H.D. Rombach. *Encyclopedia of Software Engineering*, chapter The Experience Factory. Wiley, 1994.

[123] P. Bourque and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

II

Included Papers

Appendix A

Paper A: Guiding Architectural Decisions with the Influencing Factors Method

Pia Stoll, Anders Wall
Industrial Software Systems
ABB Corporate research
pia.stoll@se.abb.com,
anders.wall@se.abb.com

Christer Norström
Computer Science and Electronics
Mälardalen University
christer.norstrom@mdh.se

In Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008,
Vancouver, BC, Canada, February, 2008

Abstract

The Influencing Factors (IF) method guides the architect through stakeholders' concerns to architectural decisions in line with current business goals. The result is a set of requirements on software quality attributes and business goals and highlighted trade-offs among software quality attributes and among business goals. The IF method is suitable for sustainable software systems since it allows new concerns, resulting from changes in business goals, stakeholder concerns, technical environment and organization, to be added to existing concerns.

A.1 Introduction

For the architect it can be difficult or even impossible to satisfy all concerns from stakeholders in one architecture. Concerns are those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, security and distribution [1]. Concerns are not always in line with stated business goals and do not always have positive impact on the product's market differentiator(s). For example the upper management wants to use standard hardware and software but this could have a negative impact on a product with high reliability as its market differentiator.

The sustainable software systems area is concerned with the architecture and designing of systems that are maintainable, supportable and modifiable during long life-times in the face of changing customer requirements and changing environments (hardware, commercial-of-the-shelf products, security threats, operating systems, communication standards). A sustainable system could have been developed according to plan, but still would not be able to cope successfully with change since the requirements for sustainable software systems will change over time. Sooner or later the systems architect has to take architectural decisions in order to satisfy changes in both business goals and changes in functional and nonfunctional requirements. In order to be pro-active the architect should continuously analyze the changing concerns and not only wait for the concerns to be transformed into functional and non-functional requirements.

Requirements usually lack any trace of what concerns they originated from and therefore it is not clear what effect they have on business goals and quality attributes. This may lead to confusion among the developers implementing requirements they do not really understand the origin of. The analyzed concerns could contribute to a more complete requirement specification than the requirement list. In [2] Nancy G. Leveson proposes an approach to provide specifications that support human problem solving and where the outcome from the concern analysis could serve as input.

It would make the architects life easier and enable better communication among diverse stakeholders if there was a time-saving method for the analyzing of concerns and their influence on the architecture. The architect could then put forward the implications of each concern and their internal relations and help the stakeholders to make rational decisions on what concerns should be prioritized. Concerns may be very stakeholder-related and it can be beneficiary

to filter the concerns so that only the facts of the concerns enter the analysis and not the relation to the stakeholders. This paper presents the Influencing Factors, IF, method which targets this challenge. The method collects concerns, extracts influencing factors from the concerns and analyzes those for their influence on business goals and software quality attributes. The result is a business goal oriented prioritization of software quality attributes.

The influencing factor is a factor that affects architecture design [3]. The influencing factors are extracted from the stakeholders' concerns in the IF method. The global analysis described in [3] analyzes each influencing factor's detailed impact on the design, the schedule and the components and not on specific business goals or software quality attributes. From a line managers concern; "We want to implement the system in Java since our developers are skilled in and enjoy working with Java" the influencing factor "Implement the system in Java" is extracted. The same influencing factor can influence different business goals and software quality attributes. The same influencing factor as above; "Implement the system in Java" can also be extracted from an upper management concern; "We want to use Java since it is too costly to re-design the system in C#". In this concern the business goal is to reduce total cost of ownership and in the first concern the business goal is to maintain jobs of the workforce on the legacy system. The method is illustrated with two industrial case studies. The first case study was performed on the upgrade of a large legacy system and the second case study on the re-factoring of an existing system. The two case study systems had a diverse set of stakeholders, such as software architect, system architect, developers, testers, product management, line management, engineers, users, and many more. Both systems suffered from an unclear understanding of what concerns were the most important. Should the architects propose architectures that try to solve all concerns, which in practice would be impossible, or should they focus on some of them?

The remainder of this paper is organized as follows; Section A.2 describes current research and the definition of business goals and software quality attributes used in the method, Section A.3 presents the relationships of enterprise, system and software architecture, Section A.4 presents the three steps of the method, Section A.5 and A.6 present two case studies where the method was applied. Section A.7 presents the conclusions of the work with the IF method and finally, future work is presented in Section A.8.

A.2 Business goals and software quality attributes

A software product interfaces with people who have an interest or share in the product's business or enterprise. These people are the stakeholders of the system. The stakeholders are users, developers, management, sales & marketing people, support engineers etc. The stakeholders experience advantages as well as disadvantages with the product depending on the concerns they find that the product should satisfy. Concerns can influence both business goals as well as system quality attributes. For example: In order to produce flexible, adaptable applications, the reflection architectural pattern, which provides a mechanism for changing structure and behavior of software systems dynamically [4], can be used. This pattern increases the modifiability of the system. But software reflection techniques may put high requirements on software developers since this way of coding is more complex than normal static programming. The complexity can lead to longer development time and therefore affect the business goal "Reduce cost of development" or "Time to Market" in a negative way. Analyzing concerns for their influence on business goals and software quality attributes is therefore necessary in order to find a suitable architectural solution. To be able to analyze the concerns influence on the business goals the business goals can be categorized. Len Bass and Rick Kazmann have categorized business goals from a number of ATAM evaluations [5]. Bass' and Kazmann's five categories are: (1) "Reduce total cost of ownership", (2) "Improve capability/quality of system", (3) "Improve market position", (4) "Support improved business processes", and (5) "Improve confidence in and perception of the system". The category "Reduce total cost of ownership" includes the subcategory "Reduce cost of development" and in the category "Improve capability/quality of system" the subcategories "Performance" and "Ease of use" are found. The categories facilitate the analysis of business goal changes with time in the IF method.

As well as having its own set of business goals a company or domain usually have its own sets of system quality attributes. In the case studies where we have applied the IF method we have used the six system quality attributes discussed in [6]; (1) Modifiability, (2) Security, (3) Usability, (4) Performance, (5) Availability and (6) Testability. In [6] it is argued that reliability is a part of the availability quality and reliability tactics are a sub set of the availability tactics. In the two case studies where the IF method was applied availability has been used to cover both availability and reliability related aspects. However, some voices have argued that reliability would have been better than availability. The architecture team or architect applying the IF method should therefore clarify

with the architecture's stakeholders what quality attributes are to be used in the beginning of the analysis.

To be able to analyze the concerns for a software product the concerns first must be extracted from the stakeholders. Interviews, document reading and personal experiences are some ways of extracting concerns. The Quality Attribute Workshop, QAW [7], is an established method to extract concerns in the form of scenarios from stakeholders in order to find prioritized business goals and software quality attributes. The QAW method can be used in conjunction with the Attribute Driven Design method, ADD [6], to achieve an architecture where all important quality attributes are considered. The QAW lets stakeholders put forward their concerns in the form of scenarios in a round-robin fashion in a one day's workshop. In order to prioritize the scenarios the stakeholders vote. This method gathers a large variety of stakeholders and let them meet and hear each others concerns. In our second case study the QAW result is incorporated with the IF method to get a complete picture of the prioritized business goals and software quality attributes.

A.3 Enterprise, System and Software Architecture

The influencing factors are part of the stakeholder concerns and include trends, technical environment, previous experiences and market demands etc, Figure 1. The stakeholder concerns can have many influencing factors.

The stakeholder concerns change over time as the influencing factors change over time. New trends, experiences and technical environments influence business goals and system quality attributes. For every change in concerns the software architect faces new business goals to satisfy, and new software quality attributes to achieve in the system respective the software architecture.

Business goals are manifested by the enterprise architecture which includes business processes and business structures, e.g. a company which sells a product needs a sales division and probably a marketing division. The enterprise architecture provides a basis for the system architecture, e.g. a company developing a safety critical software product needs a team of safety experts and processes for testing and verifying the safety properties of the product. The business goal categories presented in Section 2 are strongly related to the enterprise architecture's business processes and business structures shown in Figure 1. The first category of "Reduce total cost of ownership" means reducing cost for the entire enterprise architecture. The second, third and fifth categories; "Improve capability/quality of System", "Improve market share" and "Improve

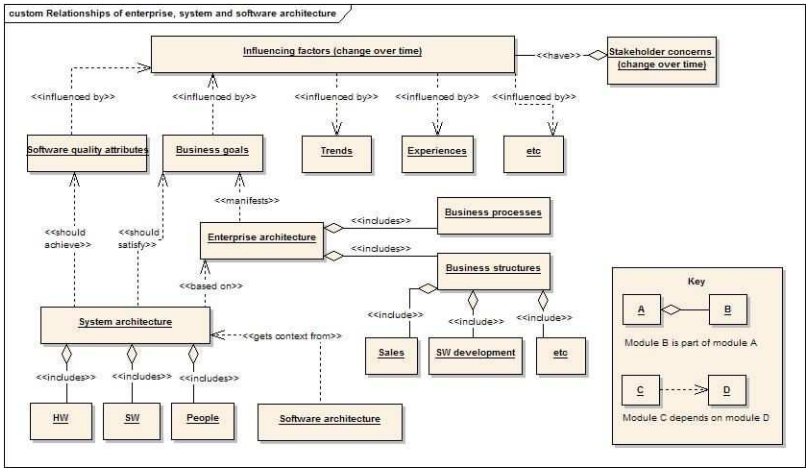

Figure 1: Relationships of enterprise, system and software architecture

confidence in and perception of system" aim at increasing the revenue for the sales and marketing part of the business structure in the enterprise architecture. "Support improved business processes" enables the software development in the business structures to run smoother.

The system architecture provides a context for the software architecture and includes beside software architecture also hardware and people.

A.4 The IF method

The Influencing Factors method consists of three steps:

- Identify influencing factors,
- Prioritize influencing factors and
- Analyze prioritized influencing factors.

Each step will be described in detail in this section. The steps are best done by the system's software architect and/or software engineer.

A.4.1 Identify influencing factors

The first step in the IF method collects concerns from different sources like: stakeholder interviews, quality attribute workshop (QAW), discussions with colleagues, search of the business related documents, and personal experiences. From the concerns influencing factors are identified. The influencing factor is a factor that affects the architecture design [3].

For identification of an influencing factor the business goal motivations and/or the system quality attribute motivation is noted (if it is stated in the concern). Not all relationships between influencing factors and system quality attributes are possible to extract from its corresponding concern. The software quality attribute influence may be more difficult to trace back to the corresponding concern than the business goal influence. A bottom-up process called affinity diagrams [8] can be used by the project team when classifying the influencing factors' effect on business goals and software attribute qualities. The team members group the influencing factors together by their software quality attribute influence in an affinity sorting process. For each software quality attribute and business goal the influence is divided into:

- Positive impact
- Negative impact
- Requires

A positive impact means an influencing factor which contributes to the achievement of the goal or the implementation of a software quality attribute. An influencing factor having a negative impact means that the influencing factor inhibits the business goal accomplishment or the software quality attribute implementation.

The influencing factor which requires a business goal or software quality attribute requires that specific tactics are used to achieve a certain quality or accomplishment of a business goal. There is a difference between having a positive impact on a system quality attribute and on having a requirement on it. For example "Support distributed development" requires high degree of "Modifiability" but has no positive impact on the quality. The influencing factor "Implement POSIX compliant software" has a positive impact on modifiability but does not require modifiability, since it is a modifiability tactic itself. The influencing factors are categorized according to their influence on business goals and software quality attributes and can then be entered into the IF matrix as shown in Figure 2. The matrix gives a good overview of the influencing factors

and especially the trade-offs between them. The influencing factor is entered into the cell which corresponds to its influence on business goals and software quality attribute. In the IF matrix for the two case studies the classified business goals as discussed in [5] and the six quality attributes from [6] are used. But it's possible to use a different set of business goals and software quality attributes that better suits the system being analyzed.

The IF matrix is one way of viewing business goal impact and software quality attribute impact of the influencing factors. If the data was put into a relational database, the user can chose different views of the data than the IF matrix in order to get the best understanding of the impact of and the internal relations between the influencing factors.

A.4.2 Prioritize influencing factors

In the first step of the IF method the influencing factors were identified and their influences on business goals and software quality attributes were documented in the IF matrix. The second step of the IF method is to identify the prioritized business goals of the system and to extract those influencing factors having a positive impact on the prioritized business goal. It can be easy to identify prioritized business goals, e.g. from interviews with upper management or from the business presentation in a Quality Attribute Workshop. However, sometimes it is more difficult to collect this information, e.g. in a distributed management organization where the system architect has little contact with the business goal responsible management. For this distributed management situation, as is shown in the first case study, it can be that the influencing factors cluster around a positive impact on a specific business goal. The architect can in this case try to verify with the management that this specific business goal is the one prioritized in the organization.

After extracting those influencing factors having a positive impact on the prioritized business goal, step three in the IF method can follow. The results should be verified with the stakeholders so that the stakeholders having concerns that are not prioritized can get an understanding of the influence of their concerns and why they are not prioritized.

A.4.3 Analyze prioritized influencing factors

After the influencing factors which have a positive impact on the prioritized business goal(s) are extracted their influence on software quality attributes can be analyzed. The factors are analyzed for their impact on the software quality

attributes. For instance, if five influencing factors have a positive impact on the prioritized business goal “Improve market share” those five factors influence on the software quality attributes are analyzed. If all of the five influencing factors require modifiability the architect knows that he/she should implement modifiability tactics and/or patterns in the architecture. This means that the architecture should try to satisfy the concerns related to the influencing factors having a positive impact on the prioritized business goal “Improve market share” and therefore apply modifiability tactics [6] and/or an architectural style [9] and/or pattern [4] with a positive impact on modifiability. If several qualities are required, techniques like the “Cost Benefit Analysis Method” [10] can be used to make cost-oriented decisions on what architectural strategy to apply. It is also important to analyze the negative impact of the prioritized factors. If the same factors requiring modifiability have a negative impact on performance the architect must take preventive measures not to get a performance drop. The IF matrix shows the internal trade-offs between business goals and internal tradeoffs between software quality attributes. It may be that influencing factors having a positive impact on the prioritized business goal “Improve market position” also have a negative impact on the business goal “Reduce total cost of ownership”. In this case the architect can discuss this with the management responsible for the business goal prioritization.

A.5 Case study 1

The first system on which the architecture team of the software system applied the IF method was a legacy system. The system suffered from an unclear understanding of what business goals the many stakeholders’ concerns were targeting and what software qualities were to be prioritized in the current development of the legacy system. For company confidentiality reasons we cannot publish all descriptions of the influencing factors. But some of the influencing factors are used as examples for clarifying purposes.

A.5.1 Identify influencing factors

The concerns from stakeholders were collected trough interviews, document reading, personal experiences and team discussions. Influencing factors were identified from the concerns and organized according to their influence on business goals and system quality attributes. For some of the factors the influence was not obvious, e.g. there was a factor having both positive and negative

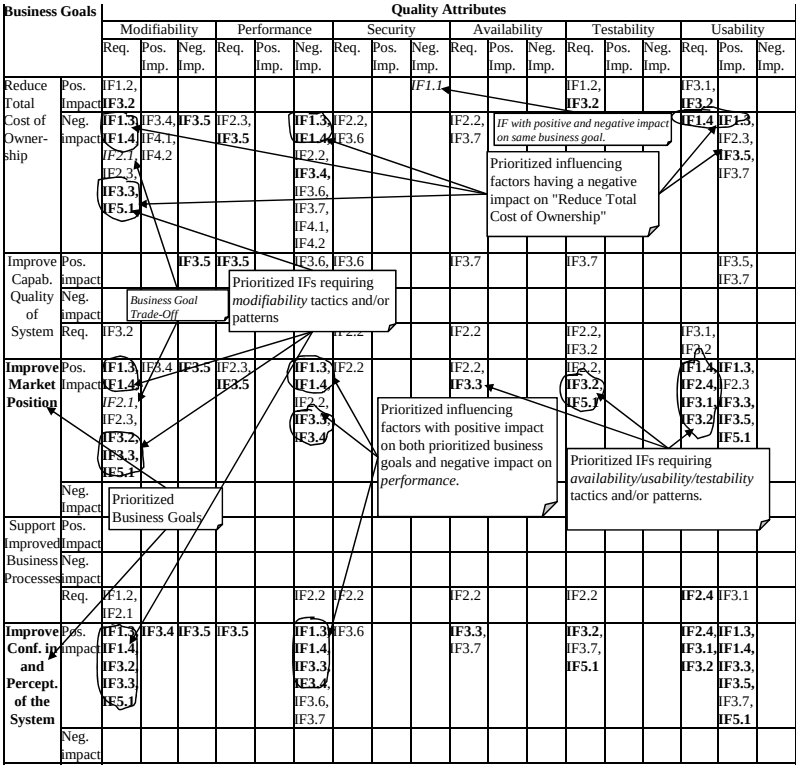

Figure 2: IF Matrix - Case Study 1

impact on the same business goal, the factor IF1.1: Microsoft Functionality Dependencies. Using ready produced functionality would make the job easier for the developers, but at the same time introduce costs in terms of licenses and dependencies on Microsoft functionality upgrades. For this factor we have both a positive and a negative impact on the same business goal “Reduce total cost of ownership”. In this particular case we argued that the license cost would be lower than the savings we would get in development cost by using the functionality. Therefore the total impact is a positive impact on the business goal and a negative impact on security due to the introduced external dependency on security patches.

We had several influencing factors having a positive impact on one business

goal and a negative impact on another business goal. Especially influencing factors having a positive impact on the business goal "Improve Market Position" tend to have a negative impact on "Reduce Total Cost of Ownership, e.g. IF2.1 "Expand geographical market to China and India". IF2.1 may involve support for new native languages which results in additional development cost. Figure 2 shows the IF matrix for the first case study.

A.5.2 Prioritize influencing factors

Since the business goal prioritization was unclear to the project team the business goal focus was clarified by analyzing the influencing factors impact on business goals. The conclusion was that a large majority of the influencing factors was focused on improving market position and confidence in, and perception of the system. We would have expected more focus on "Reduce Total Cost of Development" and "Improve Capability/Quality of System" since the legacy system was ten years old. The focus on "Improving Market Position" and "Improving Confidence in and Perception of the System" was therefore confirmed with the stakeholders. The result was that the following factors were prioritized: IF1.3, IF1.4, IF2.4, IF3.1, IF3.2, IF3.3, IF3.4 and IF5.1. They have a positive impact on both of the prioritized business goals. These factors will be analyzed in step three of the IF method.

A.5.3 Analyze prioritized influencing factors

From the matrix in Figure 2 we extracted the prioritized influencing factors and made a list of their impact on software quality attributes, Figure 3. The factors: IF1.3, IF1.4, IF3.2, IF3.3 and IF5.1 required modifiability tactics and/or patterns. The prioritized factors IF2.4, IF3.1 and IF3.3 required usability tactics and/or patterns. The factors: IF3.2 and IF5.1 required testability tactics and/or patterns and the prioritized factor IF3.3 required availability tactics and/or patterns to be implemented in the architecture. Modifiability and usability seemed therefore to be the most important software quality attributes.

Figure 3 also shows that the performance quality was negatively impacted by four of the prioritized influencing factors: IF1.3, IF1.4, IF3.3 and IF3.4. This means we had a trade-off between performance, modifiability, usability and testability.

Figure 4 shows that the business goal "Reduce total cost of development" was negatively impacted by the prioritized factors: IF1.3, IF1.4, IF3.3, IF3.4 and IF5.1. We therefore had a business goal trade-off between "Reduce Total

Impact on Software Quality Attributes		Modifiability			Performance			Security			Availability			Testability			Usability		
		Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.
Prior. Infl. Factors	IF1.3	**x**					**x**											**x**	
	IF1.4	**x**					**x**										**x**	**x**	
	IF2.4																**x**		
	IF3.1																**x**		
	IF3.2	**x**												**x**			**x**		
	IF3.3	**x**					**x**				**x**							**x**	
	IF3.4		**x**				**x**												
	IF3.5			**x**	**x**													**x**	
	IF5.1	**x**												**x**				**x**	

Figure 3: Quality Attribute Analysis Ű Case Study 1

Impact on Business Goals		Reduce Total Cost of Ownership			Improve Capability/ Quality of System			Improve Market Position			Support Improved Business Processes			Improve Confidence in and Perception of the System		
		Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.
Prioritized Influenc. Factors	IF1.3			x					x						x	
	IF1.4			x					x						x	
	IF2.4								x		x				x	
	IF3.1								x						x	
	IF3.2	x							x						x	
	IF3.3			x					x						x	
	IF3.4			x					x						x	
	IF3.5			x		x			x						x	
	IF5.1			x					x						x	

Figure 4: Business Goal Analysis Ű Case Study 1

Cost of Ownership" and "Improving Market Position"/"Improving Confidence in and Perception of the System".

A.5.4 Conclusions: Case Study 1

The business goal focus was made visible by the IF matrix. A large majority of the influencing factors had a positive impact on "Improve market position" and "Improve confidence in and perception of the system". The stakeholders confirmed this business goal focus and we could show that this focus has a strong trade-off with the business goal "Reduce total cost of ownership".

The IF matrix made internal trade-offs between business goals visible as well as internal trade-offs between software quality attributes. A side-effect of the analysis was the discovery that concerns related to the business goal category "Improve confidence in and perception of the system" was strongly correlated to the business goal category "Improve market position".

A.6 Case study 2

The system in this case study was a total reconstruction of an old system architecture that was to deliver the same features as the old system, but with additional new functions and qualities. The business goal in this case study was explicitly stated as "Increase sales to year 2009" in contrast to the first case study where the business goal was unclear. For the second case study the IF method was used to prioritize among the multitude of stakeholder concerns and legacy concerns inherited from the old system.

A.6.1 Identify influencing factors

In order to find the most important stakeholder concerns and their corresponding quality attributes a Quality Attribute Workshop [3], was held. A couple of weeks before the QAW took place we collected requirements in form of use-cases for the new system and legacy requirements from the old system. The business goal requirements were extracted from the business case presentation at the start of the QAW. The voting result was a surprise to the QAW moderators since the top-five scenarios did not include the most important legacy quality attribute requirements: performance and availability. The discussion of the result with the participating stakeholders showed that they had voted on the scenarios dealing with new features of the system and ignored the mandatory legacy features. However, by using the IF method we could extract the influencing factors from the concerns related to the legacy requirements as well.

A.6.2 Prioritize influencing factors

In this case study the prioritization of influencing factors was intensively discussed. Should we only take the ones from the Quality Attribute Workshop prioritization? By following the business goal prioritization from step two in the IF method we included both legacy requirement related influencing factors and top-five QAW scenario related influencing factors. Figure 5 show the overlap of the influencing factors in a Venn diagram. Moreover, Figure 5 also shows that important influencing factors would have been left out if only the influencing factors related to the QAW top-five scenarios would have been analyzed.

The business goal prioritization of influencing factors resulted in the prioritized influencing factors; IF1.2, IF1.2, IF1.3, IF2.1, IF2.2, IF2.3, IF2.4, IF2.5, IF2.6, IF2.7, IF3.1, IF3.2, IF3.3, IF4.1, IF4.7, IF5.1, IF5.2 and IF5.4.

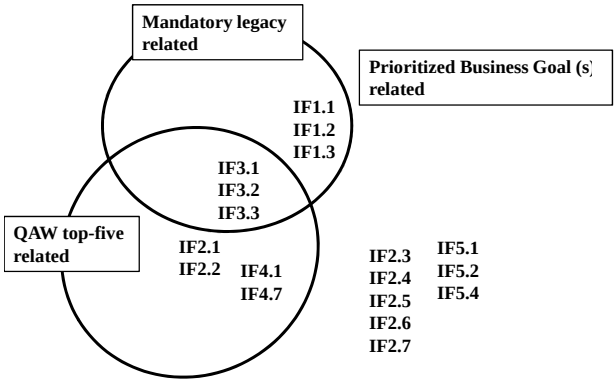

Figure 5: Overlap of influencing factors

Analyze prioritized influencing factors In step two we prioritized the influencing factors extracted from concerns related to the prioritized business goal "Improve market position".

The prioritized influencing factors have requirements on all qualities but the majority of the influencing factors require modifiability and usability, Figure 6. Most of them have a trade-off with the performance quality. The architecture can not be constructed only to satisfy the requirements on modifiability and usability without regarding the performance requirement of the IF1.2 "Implement same performance as today". The IF 1.2 has a strong legacy requirement on performance and actually drives the architecture. Figure 7 shows that nine of the eighteen prioritized influencing factors have a negative impact on the business goal "Reduce total cost of ownership". That is, the business goal "Improve Market Position" has a trade-off with the business goal "Reduce total cost of ownership".

A.6.3 Conclusions: Case Study 2

In this second case study the IF method was a necessary complement to the QAW. The stakeholders who participated in the QAW put their votes on scenarios describing new product functionality and new product qualities. Therefore all legacy requirements and prioritized business goals didn't get into the top-five scenario ranking.

One question we had regarding step two in the IF method was if it was

Impact on Software Quality Attributes			Modifiability			Performance			Security			Availability			Testability			Usability		
			Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.
Prior. Infl. Factors	Mandatory legacy req. related	IF1.1																	x	
		IF1.2			x	x														
		IF1.3					x							x						
		IF3.1	x					x						x						
		IF3.2						x					x							
		IF3.3										x								
	QAW top-five scenarios related	IF2.1																x		
		IF2.2							x									x		
		IF4.1																		
		IF4.7	x					x							x					
	Positive impact on prioritized business goals	IF2.3	x															x		
		IF2.4	x					x										x		
		IF2.5																x		
		IF2.6													x			x		
		IF2.7										x						x		
		IF5.1	x																	
		IF5.2	x																	
		IF5.4				x														

Figure 6: Software Quality Attribute Analysis Ű Case Study 2

Impact on Business Goals			Reduce Total Cost of Ownership			Improve Capability/ Quality of System			Improve Market Position			Support Improved Business Processes			Improve Confidence in and Perception of the System		
			Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.	Req.	Pos. Imp.	Neg. Imp.
Prior. Infl. Factors	Mandatory legacy req. related	IF1.1					x			x						x	
		IF1.2					x			x						x	
		IF1.3					x			x							
		IF3.1			x		x			x						x	
		IF3.2			x		x			x						x	
		IF3.3					x			x						x	
	QAW top-five scenarios related	IF2.1			x					x						x	
		IF2.2			x		x			x						x	
		IF4.1		x			x			x							
		IF4.7		x						x			x				
	Positive impact on prioritized business goals	IF2.3		x						x							
		IF2.4		x						x						x	
		IF2.5			x		x			x						x	
		IF2.6			x		x			x						x	
		IF2.7			x		x			x							
		IF5.1		x						x							
		IF5.2			x					x							
		IF5.4			x					x							

Figure 7: Business Goal Analysis Ű Case Study 2

sufficient to use prioritized business goal(s) as selection criteria for the prioritization of IFs. Case study two showed us that by using the prioritized business goal(s) as criteria we covered IFs extracted from mandatory legacy requirements concerns and from the QAW top-five scenario concerns. This could have been the case since mandatory legacy requirements are a part of the business goal "Improve market position" which has the subcategory "Expand or Retain market share". The subcategory implies that the functionality and qualities of the system must be retained or improved.

The result from the QAW was used as input to the IF method and the output from the IF method as input to the ADD method [11] and the USAP method [12].

A.7 Conclusions

The IF method extracts influencing factors from stakeholders' concerns. The influencing factor is a factor that affects the architecture design. The influencing factors' impacts on software quality attributes and business goals are analyzed. In the two case studies the gathering of concerns from stakeholders took about a person week and the contribution of each stakeholder was approximately two hours of interviews for those that couldn't participate in the one day quality attribute workshop. From the two case studies we concluded that for a skilled architect with business goals understanding and software quality attributes skills, the IF analysis of the concerns should take no longer than a day or two. Changes in business goal focus during the life-time of the software system means that new influencing factors are added. The added influencing factors and their impacts may be added to the existing ones in a relational database and the view of the impact can be presented in many ways, e.g. in the IF matrix format. In the IF method the business goal prioritization is central. It is the prioritized business goals that controls what concerns will be prioritized.

One difficulty in the IF method is the categorizing of impact on business goals and software quality attributes in step two. This is one difficulty the IF method shares with methods described in [7], [13] and [11].

A.8 Future work

We will investigate the possibility to apply the IF method before a QAW and use the result to understand the effect of the stakeholders' concerns and present

this to the stakeholders before starting the QAW. This might help the stakeholders to make more focused voting decisions.

Moreover, it would be interesting to look deeper into the correlation between business goals and software quality attributes. In our two case studies we have seen a high correlation between “Usability” and “Improve Confidence in and Perception of the System” and between “Usability” and “Improve Market Position”.

Finally more research in the field of influence of concerns on business goals and software quality attributes will make step two in the IF method more precise.

Bibliography

[1] P. Bourque and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[2] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(No. 1), 2000.

[3] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, Boston, 2000.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture A System of Patterns*, volume 1. Wiley, first edition, 1996.

[5] L. Bass and R. Kazman. Categorizing business goals for software architectures. Technical Report CMU/SEI-2005-TR-021 ESC-TR-2005-021, Software Engineering Institute, 2005.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, second edition, 2003.

[7] M. Barbacci, R. Ellison, A. Lattance, J. Stafford, C. WeinStock, and W. Wood. Quality attribute workshops, 3rd edition. Technical report, Software Engineering Institute, Pittsburgh, PA, USA, 2003.

[8] H. Beyer and K. Holtzblatt. *Contextual Design*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998.

[9] D. Garlan and M. Shaw. *Software Architecture: Perspectives on an emerging discipline*. Prentice-Hall Inc., 1996.

[10] R. Kazman, J. Asundi, and M. Klein. Making architecture design decisions: An economic approach. Technical report, Software Engineering Institute, Carnegie Mellon University, 2002.

[11] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-driven design (add), version 2.0. Technical Report CMU/SEI-2006-TR-023 ESC-TR-2006-023, Software Engineering Institute, Pittsburgh, USA, 2006.

[12] L. Bass and B. E. John. Linking usability to software architecture patterns through general scenarios. *The Journal of Systems and Software*, 66:187–197, 2003.

[13] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures, Methods and Case Studies*. Addison-Wesley, Boston, 2002.

Appendix B

Paper B: Achieving Sustainable Business for Industrial Software Systems

Pia Stoll
Industrial Software Systems
ABB Corporate research
pia.stoll@se.abb.com

Anders Wall
Industrial Software Systems
ABB Corporate research
anders.wall@se.abb.com

In Conference on Business Sustainability, Ofir, Portugal, 2008

B.1 Introduction

Sustainable development of industrial software systems with controllable outcome in terms of cost, schedule and quality despite changes originating from new technology, stakeholders' concerns, organization, and business goals during long life-times is a challenge. Unruh [1] has argued that numerous barriers to sustainability arise because today's technological systems were designed and built for permanence and reliability, not change. Sustainability is a characteristic of a process or state that can be maintained at a certain level indefinitely. The implied preference would be for systems to be productive indefinitely, to be "sustainable". For instance, "sustainable development" would be development of software systems that last indefinitely. Author Michael Pollan [2] has defined an unsustainable system simply as "a practice or process that can't go on indefinitely because it is destroying the very conditions on which it depends".

There are several factors obstructing the sustainability of the software development process:

- Competing concerns from various stakeholders affect the system and the winner among the concerns is not always the most logical. For a mature software system most probably political concerns will compete with functional concerns and affect the system.
- The system's software qualities are exposed to change, e.g. the introduction of faster multi-core processors might solve performance issues outside the scope of the architecture and therefore the focus and mission of the architecture shifts to other issues.
- The business goals of the system are exposed to change. This happens when the management shifts the focus from increase of quality to cost cut and thereby changes one important business goal for the system.
- The technical environment and organization structure change. A new platform or distributed development might be unavoidable and therefore puts requirement on change for the system.

If these factor where possible to control and a stable balance of cost, schedule, and quality outcome of the software system was achieved, the system would be a sustainable software system. The development of the software system would deliver required quality to the customers' satisfaction at the desired scheduled and cost indefinitely. However unrealistic this might seem it is

truly the goal of sustainable software development. The cost is a very important measure since a long-lived system can be achieved at a high cost but this would lead to an unsustainable development process which would eventually collapse.

Since software development is considered an art involving people and people communicating a sustainable system model must include influences from people, architecture, hardware, software, communication and unpredictable changes in form of; stakeholders' concerns' changes, technology changes, business goal changes, and organizational changes. With all influences included in one model it would be desirable to be able to predict or at least reason about the outcome of the system; cost, schedule and quality. The remaining of this paper is organized with a short overview of related work in the section "Related Research" and the issues important for sustainable industrial software systems is given in section "Issues for Sustainable Business". The paper is concluded in the section "Conclusions" followed by a short description of further work in section "Future Work".

B.2 Related Research

The importance of technical, business, and social influences on software architecture is discussed in [3] and the relationship among the technical, business, and social environments that subsequently influence future architecture is called the architecture business cycle (ABC). The ABC focuses on the creation of software architecture and the maintenance of the architecture and conformance of the system to the architecture. The ABC does not handle sustainable system issues where it's possible that the architecture has to change during the system's lifetime. An attempt to address sustainable systems can be found in [4] where the integration of established engineering methods with a development organization's life cycle is discussed. Here the Attribute Driven Design (ADD) method, [5], and the Cost Benefit Analyze Method (CBAM) [4], are suggested as means for the architect to design and chose appropriate architectural responses to the new challenges during the software development life cycle. The methods are preferably used in the development phase and the Architecture Trade-off Analysis Method (ATAM) used after the system is released and the stakeholders want to discover risks and sensitivity point in the architecture related to business goals.

For the change requests entering the system after its release the stakeholders have to take a decision if they are worth implementing or not. In an article

from Boehm [6] it is argued that software engineers should look at proposed changes to software systems as investment possibilities and calculate on the value of investing in those changes with methods similar to the methods in the investment economics, e.g. option theory. Especially the value of the success-critical stakeholders concerns should be considered important. For the sustainable software system this would mean that the software engineers have to be updated on who is a success-critical stakeholder and how to calculate the value of his/hers concern's implementation. The calculation could also serve as guidance to what concerns should be allowed to enter the system as change requests.

However calculating a correct development effort for a proposed changer request is very difficult. Joergensen [7] has showed that software project cost estimation uncertainty assessments are frequently based on expert judgment, i.e., unaided, intuition-based processes and not on formal models. His guidelines suggest, among other things, that the most promising strategies are not based on formal models, but on supporting the expert processes.

The implementation of change requests also have to have support in the development process. The process has to support unpredictable change requests as well as support their fast realization. The Scrum [8] development process has gained a lot of supporters as it's a light-weight process with a strong connection to agile development methods. Scrum considers the software development process to be a chaotic empirical process which requires close watching and control, with frequent intervention. A scrum software project is controlled by establishing, maintaining, and monitoring key control parameters. The key control parameters are backlog, issues, risk, problems and changes - task level management is not used. However in [9] Boehm argues that agile development methods are not well suited to large development organizations such as those evolving sustainable software systems. Scrum identifies the most important stakeholders and these success critical stakeholder's concerns are implemented at first. This is similar to Ruhe and Saliu [10] who describe the release planning approach based on the features' internal dependencies, the resource constraints and the stakeholders' importance.

In [11], Ziv and Richardson state the uncertainty principle of software engineering (UPSE) as "Uncertainty is inherent and inevitable in software development processes and products". The software development is described as a complex human enterprise carried out in problem domains and under circumstance that are often uncertain, vague or otherwise incomplete. The principle of uncertainty is also valid for those changes entering the development organization which are considered unpredictable in time and consequence. The control

of the sustainable software development despite the UPSE is what makes the sustainable software development challenging.

B.3 Issues for Sustainable Business

The system architecture provides a context for the software architecture and includes, beside software architecture, also hardware and people. System quality attributes and business goals influence the system architecture. The influencing factors which are factors affecting the architecture part of the stakeholder concerns [12] and include trends, technical environment, previous experiences, market demands etc.

The influencing factors change over time and hence the stakeholders' concerns change over time. The influencing factors impact and/or put requirements on system quality attributes and business goals. This leads to that the system quality attributes change as result as well as the business goals. Changing business goals can lead to changing enterprise architecture and changing development organization as business structures and business processes.

Since all these changes come from outside the software system they are uncontrollable and unforeseeable. When building software architecture from start it may be possible to build in support for foreseeable changes but not for an unforeseen change, e.g. a sudden organizational change.

B.3.1 Technology

What makes software especially difficult to develop for sustainable system is that software and hardware themselves are not sustainable. Software technologies, tools, architectures like the World Wide Web, languages like C and C# change the software engineering culture in which system builders operate and learn. In many cases the demand from the customers on smooth updates preferably in a running plant regardless of what changes occur over time translates into a requirement on backward compatibility. Backward compatibility also concerns hardware, where the customer might run the system on hardware no more available on the market.

For long-lived systems typically the components from which the system is built, have shorter life-cycles than the complete systems. Many components in a large and complex software system are acquired from third-party developers. Consequently, a system provider has no or limited control over the complete system (e.g. no access to source-code). Hence, it is very important to contin-

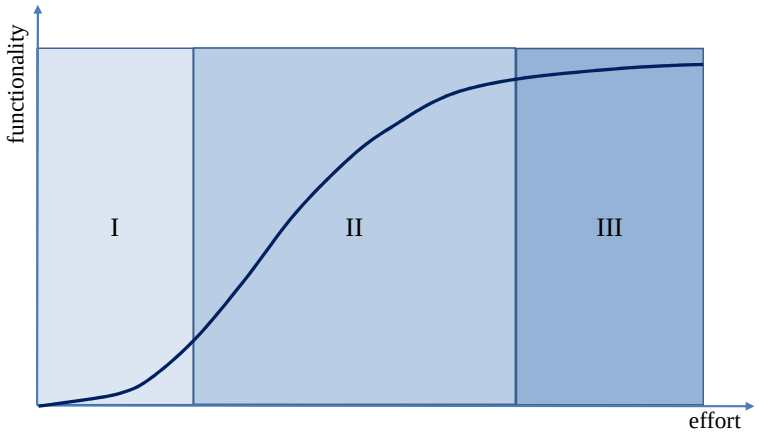

Figure 1: Product life-cycle phases

uously monitoring the sub-suppliers road-maps and to have a tight and sound relation with them. By doing so, a company have the possibility to react well in time before a particular component or technology for which the development organization has no control over gets obsolete. The fact that software technologies and commercially available software components have shorter life-cycles than what is required for the system is something that needs to be considered when designing the architecture.

Typically the life-cycle of a software product can be divided into three phases: initial design (I), evolution (II), and end-of-life (III) (see Figure 1). During the initial design phase the requirements are usually well-known and the development of new functionality requires relatively little effort. In the evolutionary phase the requirements that were not known in (I) are introduced and the effort for developing and implementing these requirements require higher effort, since consideration must be taken to what already exists in the system. The architecture developed during initial design does to a large extent define what is possible in later phases from an economical point of view.

It is important to find a balance between upfront investments in, e.g. software architectural design, and time-to-market for software development in sustainable complex industrial systems in the perspective of a product's life-cycle. By diagnosing a system's life-cycle phase in terms of trends in crucial organizational measurements we believe that it is possible to quantitatively motivate efforts in improving fundamental software qualities in order to prolong a

system's productive life-time. A typical trend in an organizational measurement could be the increasing number of person-hours invested related to the decreasing number of function points delivered. This could be an indication of a system being in the end-of-life phase (III).

Even though technology evolves in a high pace, business specific logic does not. Operating systems and hardware changes all the time but the basic principles for, e.g. control the motion of a robot, evolves slower. Another example is the paper production. The chemical process behind paper production will not change as it's defined by physical parameters and reactions. The control algorithms, which are part of the business logic, involved in controlling the pressure, strain and so on will continuously be refined but not experience major change. Usually there are great investments in the business logic and the investments are secured by intellectual property claims, so it is important to make as much as possible out of these investments. This is where we have the core competence, and the core business. Returning to the core business has proven to be successful for many companies where ABB is one of them. ABB returned its focus to automation and power distribution after some years with a broader scope. Isolating the business logic in a way that enables the technology around it to evolve with the least possible cost is crucial. The statement may seem easy enough but for researchers who have been using FORTRAN for their algorithms because its ability to process a huge amount of control parameters fast and that now have the possibility of using Matlab algorithms translated into C# just as efficiently it's not that easy. Should they now remodel the process in Matlab because in the long run C# offers more advantages than FORTRAN? What's the return of investment, the ROI, value of the change?

B.3.2 Organization

According to [3] there are three classes of organizational influences on software architecture;

- Immediate business: An organization may have an investment in certain assets, such as existing architectures and the products based on them.
- Long-term business: The architecture can form the core of the long-term infrastructure investment to meet the organization's strategic goals.
- Organizational structure: The organizational structure can shape the architecture such that the division of functionality aligns with existing units of expertise.

For sustainable systems there is a challenge in creating a sustainable architecture possible to implement under these three different organizational influences. There will be shifts in organization influence inside a development organization, e.g. if distributed development is introduced. In this case the distributed development could for instance put requirement on the architecture to support isolated module development. Another example is if the architecture suddenly has to support the migration of several products into one, as may be the case when a company acquires another company. For this case the shift in organizational structure goes from immediate business to longterm business. Development organizations often have to deal with drastic shifts like this without the customer noticing any major differences in actual system software quality.

Recognizing that change requests are something normal and that deviations from predictions will occur for a sustainable software system, the question is how to act upon them. Should a change in stakeholders' concerns toward more secure system always respond in that the system is optimized for security? Or will this be in conflict with business goals as e.g. making the system available over Internet? In traditional control theory [13], optimization theories have been developed to optimize the system parameters for stability. Something similar is needed for sustainable software systems in order to make the right system decisions in terms of economics, architecture, technology and people. There are many states that can be controlled and/or observed for a sustainable software system model:

- Software architecture- The design and the infrastructure of the system
- Software technology- The various technologies used as a technical base, such as programming environment, operating system and middle ware.
- Software components- The various proprietary and commercial components used to realize the system, examples of components are user interface, user management and transaction managers.
- Hardware- The core of the system where the software is running
- Software communication- everything regarding communication including compatibility with other vendor products, communication hardware, communication stacks and redundancy concepts.
- People interaction- Most industrial systems have people that interact with them and how this is performed is one key to the operation of the whole system.

- Development processes- Processes influence the organization and the architecture and the opposite.

The two last states, people interaction and the development processes, might be the hardest to control since they include human psychology. In [14] Berry examines programming accidents, i.e., models, methods, artifacts, and tools, to determine that each has a step that programmers find very painful and consequently avoid or postpone. The avoidance or postponement disturbs the processes in a not controllable way and leads at the worst to uncontrollable cost, schedule, and quality outcome.

But before the change request reaches the development stage it has to be approved and there is various way of handling change requirements. In [15], Erdogmus suggests a decision support theory in form of real options theory for guiding investment decisions regarding a change in the software. Typically the option theory calculations could serve as input to a change request board.

During the lifetime of a long-lived system there will be a turn-over of engineers. The engineers possess competence and know-how concerning the system. Typical examples of crucial know-how is the intention and rational behind certain architectural decisions. As engineers come and go through the organization there is a great risk that this knowledge is lost. As a consequence, poor design decision may be taken during a system's evolution which contributes to shorten the productive phase of the sustainable systems. A proper architectural documentation is one way to minimize the risk of competence drain due to turn-over of engineers. Yet again the human psychology aspect enters the field since software developers often find documentation a very painful step and avoid this as far as possible. When documenting software the people doing the documentation has to find it meaningful and ultimately, such documentation has to have some notion of intention, i.e. rationales for architectural decisions as describe by Leveson in [16].

B.3.3 Market

It's not only customers' expectations that change over time. Also a company's business goals change, e.g. penetration of new markets. Every company has its own set of business goals and to achieve a common perception of the goals, it would be beneficiary to generalize them. One approach is presented by Bass and Kazman where they have categorized the business goals from a number of ATAM evaluations [17]. Their five categories are; 1) "Reduce total cost of ownership", (2) "Improve capability/quality of system", (3) "Improve market

position", (4) "Support improved business processes", and (5) "Improve confidence in and perception of the system".

Typically there will be a movement between quality focused business goals as (1), (2), and (3) and functionality focused business goals as (3) and (5). A "fresh" software system is typically more focused on "Improve market position" and "Improve confidence in and perception of the system". New functionality is then released to customers and feedback from the release in form of change requirements and trackers leads to yet more new functionality. When the software system has grown to a certain extent the focus might shift to quality focused goals as "Reduce total cost of ownership", and "Improve capability/quality of system".

The challenge lays in balancing the shift in business goals with their interpretation to software quality goals and functionality requirements. For example "Reduce total cost of ownership" can mean outsourcing parts of the development and this puts high requirements on the modifiability and testability quality and also on software development processes different to in-house development described by Larsson et al. in [18].

Another example is the conflict of the shift towards "Reduce total cost of ownership" including the tactics to use standard hardware. If the market differentiators for the product are high robustness and backward compatibility, it means the robustness issue has to be solved with standard hardware and the backward compatibility issue with non complex architecture in order not to implement expensive development. This is truly a challenge. The customer's perception of the system should be the same, only with updated software and hardware. Industrial systems have customers running legacy hardware which have no intention or motivation to shift hardware to the latest technology. For system developers the customer's hardware puts requirement on the software to be backward compatible with the legacy hardware as well as backward compatible with legacy software.

It is not uncommon for industrial software system to have a few dominating customers who demand certain system qualities. In this case the challenge lies in to what extent the system producer can tailor the system to please one dominant customer before the other customers object to not getting their requirements met or having to pay for qualities they don't require. We have seen examples where a few dominant customers have driven a system to be too costly compared to competitors offers. The reason is that the system provides a lot of functionality which are not specifically requested by the majority of customer categories, but requires more expensive hardware infrastructure which contributes to the cost. However there is also an advantage with a large dom-

inant customer. They provide the means for the rework of one system to an extent not possible otherwise, which in the CelsiusTech case proved very successful. In the case of CelsiusTech [19], the unpredictable change in the form of the simultaneous awarding of two massive contracts (each of which was for a system beyond anything the company had ever attempted) led to a complete redesign of the system architecture based on the core assets. The new product-line architecture was the entry to new business areas not previously accessible.

B.4 Conclusions

This paper has described the challenges for the development of sustainable industrial software systems. The most important factor to recognize is the factor of time and its effect on system development since industrial software systems often have long lifetimes. The second factor to recognize is that change in organization, technology, and market over time is something inevitable and that the development has to calculate for this. The third factor to recognize is that changes are not always predictable or foreseeable and that a static system could have difficulties to host unpredictable and unforeseeable changes. The forth factor to recognize for industrial systems is that their customers most often don't want to experience any change since a change requiring knowledge update or process interruptions is costly. The last factor to recognize is that the producer can achieve the desired quality and cost despite unpredictable changes at an unreasonable cost, but this would lead to an unsustainable development process which would eventually collapse. This leads us to the conclusion that the sustainable industrial software system has to control the cost, quality, and schedule outcome of the system despite unpredictable and predictable changes in organization, market, and technology affecting the system over time.

B.5 Future Work

Future work will include an attempt to establish a sustainable software system model, including measures for the key states important for the control of the outcome of a sustainable industrial software system. In this work software economics will be a key essence influencing the software engineering theory for the model.

Bibliography

[1] G.C Unruh. Escaping carbon lock-in. *Energy Policy*, vol. 30(no.4):pp. 317–325, 2002.

[2] P. Pollan. Our decrepit food factories. *New York Times*, 2007.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, second edition, 2003.

[4] R. Kazman, J. Asundi, and M. Klein. Making architecture design decisions: An economic approach. Technical report, Software Engineering Institute, Carnegie Mellon University, 2002.

[5] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-driven design (add), version 2.0. Technical Report CMU/SEI-2006-TR-023 ESC-TR-2006-023, Software Engineering Institute, Pittsburgh, USA, 2006.

[6] B.W. Boehm and K.J. Sullivan. Software economics: a roadmap, 2000.

[7] M. Joergensen. Evidence-bases guidelines for assessment of software development cost uncertainty. *IEEE transactions on software engineering*, 31, 2005.

[8] K. Schwaber. Scrum development process. Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger 6(4), October 1995.

[9] B. W. Boehm. A view of 20th and 21st century software engineering., 2006.

[10] G. Ruhe and M.O. Saliu. The art and science of software release planning. *IEEE Software*, 22:47–53, 2005.

[11] H. Ziv and D.J Richardson. The Uncertainty Principle in Software Engineering. In *19th International Conference on Software Engineering (ICSE'97)*, 1997.

[12] P. Stoll, A. Wall, and C. Norström. Guiding Architectural Decisions with the Influencing Factors Method. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008, 2008.

[13] L. Ljung. *System Identification - Theory For the User*. Prentice Hall, Upper Saddle River, N.Y, 1999.

[14] D.M Berry. The inevitable pain of software development: Why there is no silver bullet. In *LNCS 2941*. Springer Verlag, 2004.

[15] H. Erdogmus. Valuation of complex options in software development, 1999.

[16] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(No. 1), 2000.

[17] L. Bass and R. Kazman. Categorizing business goals for software architectures. Technical Report CMU/SEI-2005-TR-021 ESC-TR-2005-021, Software Engineering Institute, 2005.

[18] S. Larsson, A. Wall, and P. Wallin. Assessing the influence on processes when evolving the software architecture, 2007.

[19] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

Appendix C

Paper C: Preparing Usability Supporting Architectural Patterns for Industrial Use

Pia Stoll
ABB Corporate Research
Forskargr änd 6
SE 72178 Västerås, Sweden
Tel: +46 21 32 30 00
pia.stoll@se.abb.com

Len Bass, Bonnie E. John, Elspeth Golden
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA, USA 15213
Tel: 1+412 268 2000
bej@cs.cmu.edu, ljb@sei.cmu.edu,
egolden@cmu.edu

In International Workshop on the Interplay between Usability Evaluation and Software Development, I-USED 2008, CEUR Workshop proceedings series, ISSN 1613-0073, Pisa, Italy, September 24th, 2008

Abstract

Usability supporting architectural patterns (USAPs) have been shown to provide developers with useful guidance for producing a software architecture design that supports usability in a laboratory setting [1]. In close collaboration between researchers and software developers in the real world, the concepts were proven useful [2]. However, this process does not scale to industrial development efforts. In particular, development teams need to be able to use USAPs while being distributed world-wide. USAPs also must support legacy or already partially-designed architectures, and when using multiple USAPs there could be a potentially overwhelming amount of information given to the software architects. In this paper, we describe the restructuring of USAPs using a pattern language to simplify the development and use of multiple USAPs. We also describe a delivery mechanism that is suitable for industrial-scale adoption of USAPs. The delivery mechanism involves organizing responsibilities into a hierarchy, utilizing a checklist to ensure responsibilities have been considered, and grouping responsibilities in a fashion that both supports use of multiple USAPs simultaneously and also points out reuse possibilities to the architect.

Categories and Subject Descriptors

D.2.2 { **Design Tools and Techniques**} User interfaces; D.2.11 { **Software Architectures**} : Patterns; H.5.2 {**User Interfaces**} Theory and Methods

General Terms

Design, Human Factors.

Keywords

Software Architecture, Usability, Human-Computer Interaction, HCI

C.1 Introduction

The Software Engineering community has recognized that usability affects not only the design of user interfaces but software system development as a whole. In particular, efforts are focused on explaining the implications of usability on software architecture design ([3], [4], [5], [6], [7]). One effort in this area is to produce artifacts that communicate usability requirements in a form that can be effectively used for software architecture evaluation and design. These usability supporting architectural patterns (USAPS) have been shown to improve the ability of software architects to design higher quality architectures to support usability features such as the ability to cancel a long-running command ([1], [8]). Other uses of USAPs in industrial settings have been productive [2] but have revealed some problems that prevent scaling USAPs to widespread industrial use. These problems include:

1. Prior efforts have involved personal contact with USAP researchers, either face to face or in telephone conversations. This process does not scale to widespread industrial use.

2. The original USAPs included UML diagrams modifying the MVC architectural pattern to better support the usability concern. Although intended to be illustrative, not prescriptive, software architects using other overarching patterns (e.g., legacy systems, SOA) viewed these UML diagrams as either unrelated to their work or as an unwanted recommendation to totally redesign their architecture.

3. The original use of USAPs was as a collection of individual patterns. Even using one pattern involved processing a large amount of detailed information. Applying multiple USAPs simultaneously is likely to overwhelm software architects with information.

In this paper, we introduce a pattern language [9] for USAPs that reduces the information that architects must absorb and produces information at a level that applies to all architectures. We also discuss the design of a delivery mechanism suitable for industrial scale adoption of USAPs.

C.2 Background

A USAP has six types of information. We illustrate the types with information from the cancellation USAP [10]:

1. A brief scenario that describes the situation that the USAP is intended to solve. For example, “The user issues a command then changes his or her mind, wanting to stop the operation and return the software to its pre-operation state.”

2. A description of the conditions under which the USAP is relevant. For example, “A user is working in a system where the software has long-running commands, i.e., more than one second.”

3. A characterization of the user benefits from implementing the USAP. For example, “Cancel reduces the impact of routine user errors (slips) by allowing users to revoke accidental commands and return to their task faster than waiting for the erroneous command to complete.”

4. A description of the forces that impact the solution. For example, “No one can predict when the users will want to cancel commands”

5. An implementation-independent description of the solution, i.e., responsibilities of the software. For example, one implication of the force given above is the responsibility that “The software must always listen for the cancel command.”

6. A sample solution using UML diagrams. These diagrams were intended to be illustrative, not prescriptive, and are, by necessity, in terms of an overarching architectural pattern (e.g., MVC).

It is useful to distinguish USAPs from other patterns for software design and implementation. USAPs are not user interface patterns, that is, they do not represent an organization or look-and-feel of a user interface e.g., [11]; they are software architecture patterns that support UI concerns. Nor are USAPs structural software architecture patterns like MVC; they are patterns of software responsibilities that can be applied to any structure. As such, they can be applied to any legacy architecture and can support the functionality called for in UI patterns.

C.3 A Pattern Language for USAPs

Through collaboration among academic and industrial researchers and usability engineers, we are constructing three USAPs for process control and robotics applications. The first author and her colleagues in the industry research team

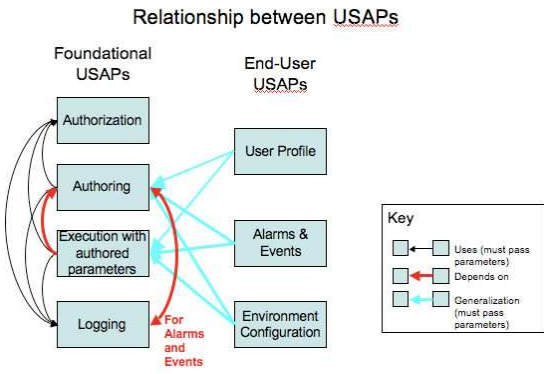

Figure 1: USAP Pattern Language for "User Profile", "Alarms, Events and Alerts", and "Environment Configuration"

drafted an "Alarms, Events and Alerts" USAP while, independently, the last three authors drafted a "User Profile" USAP and an "Environment Configuration" USAP. When these three USAPs were considered together, it was clear that there was an enormous amount of redundancy in the responsibilities necessary for a good solution. In addition, in preliminary discussions, industry software architects reacted negatively to the large amount of information required to implement any one of the USAPs.

Our early work [4] recognized the possibility of reusing such software tactics as separating authoring from execution and recording (logging), but our subsequent work had not incorporated that notion, treating each USAP as a separate pattern. A consequence of focusing on industrial use is that reuse in constructing and using USAPs was no longer an academic thought experiment, but a necessity if industrial users are to construct and use USAPs themselves.

We observed that both the industry research team and the academic research team independently grouped their responsibilities into very similar categories. This led us to construct a pattern language [2] that defines relationships between USAPs with potentially reusable sets of responsibilities that can lead to potentially reusable code. Our pattern language relating "Alarms, Events and Alerts", "User Profile" and "Environment Configuration" is shown in Figure 1.

The pattern language has two types of USAPs. "End-User USAPs" follow the structure given in Section C.4. Their purpose from a user's point of view can be expressed in a small scenario, they have conditions under which they are relevant, benefits for the user can be expressed and they require the fulfillment

of software responsibilities in the architecture design. End-User USAPs are used by the requirements team to determine which are applicable to the system being developed. In this example, they are "User Profile", "Alarms, Events and Alerts", and "Environment Configuration".

The pattern language also contains what we are calling "Foundational USAPs". These do not have the same six portions as the End-User USAPS. For example, there is no scenario, no description of conditions, and no benefits to the user for the Foundational USAPs. Rather, they act as a framework to support the construction of the End-User USAPs that make direct contact to user scenarios and usability benefits. For example, all of the End-User USAPs that we present have an authoring portion and an execution portion, that is, they are specializations of the Authoring Foundational USAP and the Execution with Authored Parameters Foundational USAP. These foundational USAPs make use of other foundational USAPs, Authorization and Logging. We abstracted the commonalities of the End-User USAPs to derive the responsibilities of the Foundational USAPs. The responsibilities in the Foundational USAPs are parameterized, where the parameters reflect those aspects of the End-User USAPs that differ.

An example of the parameterization is that the Authoring Foundational USAP and the Execution with Authored Parameters Foundational USAP each have a parameter called SPECIFICATION. The value of SPECIFICATION is "Conditions for Alarm, Event and Alerts", "User profile", and "Configuration description" for the three End-User USAPs, respectively. In addition to parameterization, End-User USAPs explicitly list assumptions about decisions the development team must make prior to implementing the responsibilities. For example, in the "Alarms, Events and Alerts" End-User USAP, the development team must define the syntax and semantics for the conditions that will trigger alarms, events or alerts. End-User USAPs may also have additional responsibilities beyond those of the Foundational USAPs they use. For example, the "Alarms, Events and Alerts" End-User USAP has an additional responsibility that the system must have the ability to translate the names/ids of externally generated signals (e.g., from a sensor) into the defined concepts. Both the assumptions and additional responsibilities will differ for the different End-User USAPs.

There are three types of relationships among the Foundational USAPs and these are shown in Figure 1 as different color arrows. The Generalization relationship (turquoise) says that the Foundational USAP is a generalization of part of the End-User USAP. The End-User USAP passes parameters to that Foundational USAP and, if there are any conditionals in the responsibilities of the

Foundational USAP, the End-User USAP may define the values of those conditionals. The Uses relationship (black) also passes parameters, but the USAPs are at the same level of abstraction (the foundational level). The Depends-On relationship (red) implies a temporal relationship. For example, the system cannot execute with authored parameters unless those parameters have first been authored. The double headed arrow between authoring and logging reflects the possibility that the items being logged may be authored and the possibility that the identity of the author of some items may be logged.

Foundational USAPs each have a manageable set of responsibilities (Authorization has 11; Authoring, 12; Execution with authored parameters, 9; and Logging 5), as opposed to the 21 responsibilities of the Cancel USAP that seemed to be too much for our experiment participants to absorb in one sitting [1]. These responsibilities are further divided into groups for ease of understanding, e.g., Authoring is separated into Create, Save, Modify, Delete and Exit the authoring system. This division into manageable Foundational USAPs simplifies the creation of future USAPs that use them. For example, the User Profile End-User USAP requires only the definition of parameters and the values for one conditional, and pointers to the Authoring and Execution Foundational USAPs.

C.4 Delivering a single USAP to Software Architects

The roadblocks to widespread use of USAPs in industry identified in the introduction were (1) the need for contact with USAP researchers in the development process, (2) reactions to examples using a particular overarching architectural pattern (MVC) and (3) an overwhelming amount of information delivered to the software architect. Data from our laboratory study and the pattern language outlined above put us in a position to solve these problems. Our laboratory study [1] showed that a paper-based USAP could be used by software engineers[1] without researcher intervention, to significantly improve their design of an architecture to support the users' need to cancel long-running commands. Although significantly better than without a USAP, these software engineers seemed to disregard many of the responsibilities listed in the USAP in their designs. To enhance attention to all responsibilities, we have

[1] The participants in our lab study had a Masters in SE or IT, were trained in software architecture design, and had an average of over 21 months in industry.

chosen to design a web-based system that presents responsibilities in an interactive checklist (Figure 2). The design includes a set of radio buttons for each responsibility that are initially set to "Not yet considered." The architect reads each responsibility and determines whether it is not applicable to the system being designed, already accounted for in the architecture, or that the architecture must be modified to fulfill the responsibility. If "Not applicable", "Must modify architecture to address this" or "Architecture addresses this" is selected, then the responsibility's check-box is automatically checked. If "Not considered", "Must modify architecture or "Discuss status of responsibility", is selected, the responsibility will be recorded in To-Do list generated from the website (Figure 3). We expect this lightweight reminder to consider each and every responsibility will not be too much of a burden for the architect, but will increase the coverage of responsibilities, which is correlated with the quality of the architecture solution [8].

As Figure 2 show, the responsibilities are arranged in a hierarchy, which reflects both the relationship of End-User and Foundational USAPs and the internal structure within a Foundational USAP. This hierarchy divides the responsibilities into manageable subparts. The check-boxes enforce this structure by automatically checking off a higher-level box when all its children have been checked off, and conversely, not allowing a higher-level box to be checked when one or more of its children are not. Thus, this mechanism simultaneously addresses the problems of providing guidance without intervention by USAP researchers and simplifying the information provided to the software architect. Another mechanism for simplifying the information delivered to an architect is that each responsibility has additional details available only by request of the architect. These details include more explanation, rationale about the need for the responsibility and the forces that generated it, and some implementation details. This information is easily available, but not "in the face" of the software architect. As well as simplifying the presentation, this mechanism deemphasizes the role of illustrative examples situated in reference architecture like MVC. We expect that this presentation decision will reduce the negative reactions to generic example UML diagrams. When using the tool in-house in industry, the reference architecture used in example solutions could be changed to an architecture used by that industry. This would both accelerate understanding of the examples and increase the possibility of re-using the sample solution. This presumes that the tool is constantly managed and updated by in-house usability experts and software architects, a presumption facilitated by delivering the examples in separate web pages.

Although the hierarchy of responsibilities reflects the relationship of the

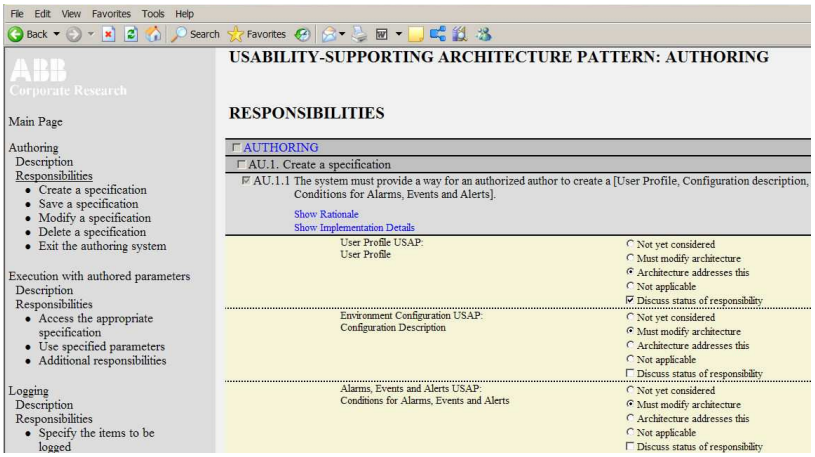

Figure 2: Prototype of a web-based interface for delivering USAP responsibilities to industry software architects

End-User USAPs and the Foundational USAPs, the difference between the types of USAPs is not evident in the presentation of responsibilities. It was a deliberate design choice to express each responsibility in terms of the End-User USAP's vocabulary. Thus, the responsibilities in Figure 2 are couched in terms of "User Profile", "Configuration Description", "Conditions for Alarms, Events, and Alerts" and this string replaces the parameter SPECIFICATION in the Foundational Authoring USAP.

In the next section, we discuss how we anticipate managing the situation when the architect chooses multiple USAPs as being relevant to the system under construction. This will allow distribute architecture teams both to record rationale for their choice and to discuss potential solutions. Attaching design rationale and discussion is optional so our delivery tool will support discussion, but not require it, keeping the tool lightweight.

At any point in the process of considering the different responsibilities, the architect can generate a "to do" list. This is a list of all of the responsibilities that have been checked as "Not yet considered" or "Must modify architecture". See Figure 3 for an example. The list can then be entered into the architect's normal task list and will be considered as other tasks are considered.

Supporting world wide distribution of the architecture team in the use of USAPs has two facets.

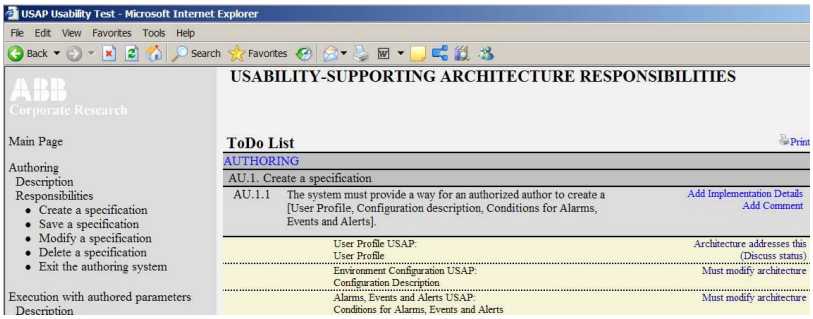

Figure 3: Prototype “to do” list produced from those responsibilities that are marked as requiring architectural modification

- Enable world wide access
- Reduce the problems associated with simultaneous updates by different members of the team.

The use of the World Wide Web for delivery allows world wide access with appropriate access control. Standard browsers support the concept of check lists and producing the “to do” lists.

Allowing simultaneous updates is not supported by standard browsers. Some Wikis do support simultaneous updates, e.g. MediaWiki[2], but we do not yet know whether these wikis directly support checklists and the generation of “to do” lists. We are currently investigating which tool or combination of tools will be adequate for our needs and what modifications might have to be made to those tools.

C.5 Delivering multiple USAPs to software architects

Our motivation for developing the USAP Pattern Language was partially to simplify the delivery of USAPs when multiple USAPs are relevant to a particular system. We also want to indicate to the architect the possibilities for reuse. In this section, we describe how we anticipate accomplishing these two goals.

[2]www.mediawiki.org

Recall that the Foundational USAPs are parameterized and each End User USAP provides a string that is used to replace the parameter. For instance, consider a responsibility from the Authoring Foundational USAP "The system must provide a way for an authorized user to create a SPECIFICATION". When three End User USAPs are relevant to the system under design, such as "User Profile", "Environment Configuration", and "Alarms, Events and Alerts", the three responsibilities are displayed to the architect as "The system must provide a way for an authorized user to create a [User Profile, Configuration description, Conditions for Alarm, Event and Alerts]". This presentation satisfies two goals and introduces one problem. Presenting three responsibilities as one reduces the amount of information displayed to the architect since every Foundational USAP responsibility is displayed only once, albeit with multiple pieces of information. This presentation also indicates to the architect the similarity of these three responsibilities and hence the reuse possibilities of fulfilling them through a single piece of parameterized code.

The problem introduced by this form of the presentation is that now the radio buttons becomes ambiguous. Does the entry "Architecture addresses this" mean that all of the three responsibilities have been addressed or only some of them? We resolve this ambiguity by repeating the radio buttons three times, once for each occurrence of the responsibility. Thus, the three responsibilities will be combined into one textual description of the responsibility but three occurrences of the radio buttons.

C.6 Current status and future work

At this writing, we have developed the pattern language for three End User USAPs and four Foundational USAPs (Figure 1) and have fleshed out all the responsibilities for these seven USAPs. We have constructed a prototype delivery tools for a browser based checklist and "to do" list generator. We plan to test the delivery mechanism in an ongoing industrial development effort. This will demonstrate strengths and weaknesses of our approach and we will iterate to resolve any problems or capitalize on any opportunities. One suggestion put forth in early industry feedback is to enhance the to-do list by assigning expected effort to each responsibility. One requirements engineer at ABB said that her perception of the effort needed to implement a scenario had been thoroughly revised just be looking at the to-do list. By adding estimated hours to the responsibilities, industry would get a better estimate of the usability improvements' translation into software implementation cost. These estimates

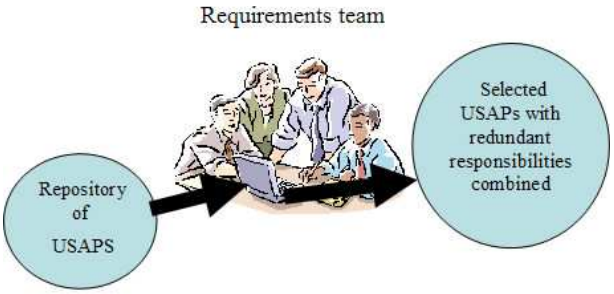

Figure 4: Tool to support the requirements elicitation process

would vary depending on many factors such as underlying architectural style, implementation language, skill of programmers, etc. but a large organization may have enough data from previous projects to make such estimates for their organization. In addition, such a feature could emphasize the savings realized by reuse; responsibility-implementations that serve multiple End-User USAPs would show up as requiring very little effort after the first implementation.

The delivery platform that we have described here, to be used by software architects, is envisioned to be the final portion of a tool chain. There are two additional roles involved in the development and use of USAPs. First, USAP developers will have to create USAPs within the stylized context of the USAP Pattern Language. Tool support for USAP developers will greatly simplify the creation of USAPs.

The second role is the requirements definers; often a team comprised of technologists and human factors engineers, usability engineers, designers, or other users or user advocates. Figure 4 shows how we envision a tool supporting this role.

The requirements team has available to them a repository of USAPs. They select the ones that are appropriate for the system being constructed. In our experience, the USAP end-user scenarios are very general and can be used to invoke ideas about how they apply to the system at hand. However, industrial teams would like to tailor these scenarios to match their everyday usability issues. Thus, the tool supporting requirements definers will allow them to rewrite the general scenarios to suit their specific application.

The tool then creates input for the delivery tool while simultaneously combining redundant responsibilities. The output of the requirements definition process will then be presented to software architects, as described in this pa-

per, to aid in their architecture design process.

In summary, USAPs have been proven to be useful to software architects but have also demonstrated some problems that hinder industrial use. Definition of a USAP Pattern Language and an appropriate selection of tools supporting the roles involved in the creation and use of USAPs should simplify industrial use. We are currently constructing versions of these tools and testing the extent to which they do, in fact, enable the industrial use of USAPs.

C.7 Acknowledgments

We would like to thank Fredrik Alfredsson and Sara Lövemark for their contributions to the “Alarms, Events and Alerts” USAP. This work was supported in part by funds from ABB Inc. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ABB.

Bibliography

[1] E. Golden, B. E. John, and L. Bass. The value of a usability-supporting architectural pattern in software architecture design: A controlled experiment. In *Proceedings of the 27th International Conference on Software Engineering, ICSE*, St. Louis, Missouri, May 2005.

[2] R. J. Adams, L. Bass, and B. E. John. *Applying general usability scenarios to the design of the software architecture of a collaborative workspace.* In A. Seffah, J. Gulliksen and M. Desmarais (Eds.) Human-Centered Software Engineering: Frameworks for HCI/HCD and Software Engineering Integration. Kluwer Academic Publishers, 2005.

[3] L. Bass, B. E. John, and J. Kates. Achieving usability through software architecture. Technical Report No. SEI-TR-2001-005, Carnegie Mellon University/Software Engineering Institute, Pittsburgh, PA, 2001.

[4] L. Bass and B. E. John. Linking usability to software architecture patterns through general scenarios. *The Journal of Systems and Software*, 66:187–197, 2003.

[5] E. Folmer. *Software Architecture Analysis of Usability*. PhD thesis, Department of Computer Science, University of Groningen, Groningen., 2005.

[6] E. Folmer, J. van Gurp, and J. Bosch. A Framework for capturing the Relationship between Usability and Software Architecture. Software Process: Improvement and Practice, Volume 8, Issue 2. Pages 67-87., 2003.

[7] N. Juristo, A.M. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *Software Engineering, IEEE Transactions on*, 33(11):744–758, Nov. 2007.

[8] E. Golden, B. E. John, and L. Bass. Quality vs. quantity: Comparing evaluation methods in a usability-focused software architecture modification task. In *Proceedings of the 4th International Symposium on Empirical Software Engineering*, Noosa Heads, Australia, November 17-18 2005.

[9] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA, 1977.

[10] B. E. John, L. Bass, M.-I. Sanchez-Segura, and R.J. Adams. Bringing usability concerns to the design of software architecture. In *Proceedings of The 9th IFIP Working Conference on Engineering for Human-Computer Interaction and the 11th International Workshop on Design, Specification and Verification of Interactive Systems*, Hamburg, Germany, 2004.

[11] J. Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media: Sebastopol, CA, 2006.

Appendix D

Paper D: Supporting Usability in Product Line Architectures

Pia Stoll
ABB Corporate Research
Forskargr änd 6
SE 72178 Västerås, Sweden
Tel: +46 21 32 30 00
pia.stoll@se.abb.com

Len Bass, Bonnie E. John, Elspeth Golden
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA, USA 15213
Tel: 1+412 268 2000
bej@cs.cmu.edu, ljb@sei.cmu.edu,
egolden@cmu.edu

In Software Product Lines Conference, SPLC 2009, San Francisco, USA, August, 2009

Abstract

This paper addresses the problem of supporting usability in the early stages of a product line architecture design. The product line used as an example is intended to support a variety of different products each with a radically different user interface. The development cycles for new products varies between three years and five years and usability is valued as an important quality attribute for each product in the line.

Traditionally, usability is achieved in a product by designing according to specific usability guidelines, and then performing user tests. User interface design can be performed separately from software architecture design and prototyping, but user tests cannot be performed before detailed UI design and prototyping. If the user tests discover usability problems leading to required architectural changes, the company would not know about this until two years after the architecture design was complete. This problem was addressed by identifying a collection of 19 well known usability scenarios that require architectural support. In our example, the stakeholders for the product line prioritized three of these scenarios as key product-line scenarios for improving usability. For each of these three chosen product-line scenarios we developed an architectural responsibility pattern that provided support for the scenario. The responsibilities are expressed in terms of architectural requirements with implementation details and rationales. The responsibilities were embodied in a web based tool for the architects.

The two architects for the product line developed a preliminary design and then reviewed their design against the responsibilities supporting the scenarios. The process of review took a day and the architects conservatively estimated that it saved them five weeks of effort later in the project.

D.1 Introduction

ABB, a global leader in power and automation technologies, provides systems that enable utility and industry customers to improve their performance while lowering environmental impact. To that end, ABB must design and implement extensive long-lived software systems. This paper presents the results of a collaboration between ABB Corporate Research, ABB core business units, and Carnegie Mellon's Software Engineering Institute and Human-Computer Interaction Institute to support usability within the context of a product line architecture being newly developed.

The best method to support usability concerns through software architecture has been the subject of some investigation over the past years. In addition to the authors' work ([1] , [2], [3]), Folmer and his colleagues ([4], [5]) and Juristo and her colleagues [6] have investigated the relationship between software architecture and usability. None of this work has gained widespread industrial acceptance primarily because all of the results reported require the hands-on involvement of the researchers. Our goal in the project reported on here was to deliver appropriate knowledge concerning usability and software architecture to ABB's software architects in a format and at a time that would benefit their design, in a way that could scale to worldwide development efforts.

This paper reports the results of a new approach to providing usability knowledge to software architects early in the design process and without the active participation of the researchers. The activities reported on include

- Stakeholders selecting several usability scenarios important to the project under design
- The research team defining architectural patterns to satisfy the scenarios chosen
- The research team embedding those patterns into a tool
- The architects using the tool for one day to review an early version of their design. They did this without previous exposure to the patterns and without any participation by the research team.
- The architects reflecting on the impact of their use of the tool. They estimated that it saved them five weeks of work.

It was the next to last bullet Ű the architects using the knowledge embedded in the tool Ű that can be scaled. Since the tool is web based, architects in any

project for which the usability scenarios embedded in the tool are relevant can use the tool and the knowledge embedded in it without any involvement of the researchers involved.

D.2 Background

Prior to the collaboration reported in this paper, the project team in an ABB business unit developing a new product line of systems, together with an ABB research team, had done a use case analysis, performed a Quality Attribute Workshop to collect non-functional requirements from prioritized scenarios [7], used the Influencing Factors method [8] and conducted the first step of the Product Line Architecture development approach [9] with the identification of commonalities and variation points. Thus, from the requirements collection and analysis perspective, the project team was well prepared when they began to outline the architecture. The software architects had just starting sketching the architecture and had not yet written any code. Their implementation plan started with the backbone of the product line system, the core functionality, which would support all the variation points for the products. Usability had been prioritized as one of three most important software qualities for the new architecture during the Quality Attribute Workshop. One of the challenges for this project therefore was how to incorporate usability requirements into the core architecture early without having either a designed user interface or a finished prototype for user tests. The user interfaces are to be developed individually for each product and each product will use common core parts of the system. The product development cycles will vary between three and five years. Thus, the question was: How can we best support usability early when the product prototypes cannot be user tested until years after the architecture design is to be completed? Most of standard usability evaluation techniques Ű questionnaires, heuristic evaluation, think-aloud usability studies Ű depend on having at least a paper prototype if not a running system. These types of tests may find modifications whose satisfaction requires changing the architecture. The effort of re-working the product-line architecture and the design for a line of products two years or even four years after the architecture has been established would be tremendous. The risk of finding severe usability problems requiring architectural work late in this development cycle was not acceptable and ABB decided to use usability supporting architectural patterns (USAPs) in a collaboration with CMU. The decision was based on the fact that USAPs use generic usability scenarios common in complex systems and from

these construct generic software architecture responsibilities. By working this way ABB expected to support some of the major usability issues early in the software design phase without having an actual user interface design in place.

A USAP is, as the name suggests, a software architectural pattern that provides instructions as to how to achieve specific usability scenarios. These patterns are at the level of software architecture responsibilities. Examples of such patterns are canceling a long-running command, aggregating data, or supporting personalization of the user interface. Note that these are software architecture patterns in the flavor of [10] not usability patterns such as in [11]. Usability patterns describe user interface patterns such as an organization's look and feel whereas software architecture patterns suggest software design solutions to specific problems.

As originally conceived, a USAP included six types of information. We illustrate the types with information from the cancellation USAP [3].

1. A brief scenario that describes the situation that the USAP is intended to solve. For example, "The user issues a command then changes his or her mind, wanting to stop the operation and return the software to its pre-operation state."

2. A description of the conditions under which the USAP is relevant. For example, "A user is working in a system where the software has long-running commands, i.e., more than one second."

3. A characterization of benefits to the user from implementing the USAP. For example, "Cancel reduces the impact of routine user errors (slips) by allowing users to revoke accidental commands and return to their task faster than waiting for the erroneous command to complete."

4. A description of the forces that impact the solution. For example, "No one can predict when the users will want to cancel commands"

5. An implementation-independent description of the solution, i.e., responsibilities of the software. For example, one implication of the force given above is the responsibility that "The software must always listen for the cancel command."

6. A sample solution using UML-style diagrams. These diagrams were intended to be illustrative, not prescriptive, and were, by necessity, in terms of an overarching architectural pattern (e.g., MVC).

USAPs have been shown to significantly improve a software architecture design in laboratory experiments [2]. They have also been used in real development settings, with heavy involvement from the developers of the USAP [1]. However, these prior uses of USAPs suffer from two defects. First, the industrial usages have all involved the developers of USAPs. This clearly does not scale up. Secondly, the laboratory experiments were paper-based and the participants omitted important responsibilities of the USAPs, leaving additional room for quality improvement. Our initial goals when we considered applying USAPs to the ABB project were to solve the two major problems that we have discussed.

1. The designers should be able to utilize the USAPs without immediate researcher involvement.
2. The designers should be encouraged to consider all of the responsibilities.

D.3 Prior work

Prior to working with ABB, the last three authors performed a laboratory experiment to test the utility of the various types of information in a USAP. The results also suggested directions for a delivery tool for USAPs, so summarizing the experiment and results here sets a context for the experience reported in this paper.

There were three different conditions in the experiment. Participants in the first condition were given only the scenario that describes the situation that the USAP is intended to solve. This mimics a common relationship between usability engineers and software designers in that the usability engineers provide general requirements (e.g., the system must be able to cancel long-running commands) but the creation of a design solution to fulfill those requirements is up to the software engineers.

Participants in the second condition were provided with the scenario plus a list of responsibilities that may have to be fulfilled to satisfy the scenario, depending on the particular system to which the scenario is being applied. Participants in the third condition were provided with the scenario, the list of responsibilities, and a sample solution using the MVC overarching architecture pattern, expressed in UML-style diagrams.

The results of the experiment were that providing the participants with information about responsibilities and a sample solution resulted in significantly

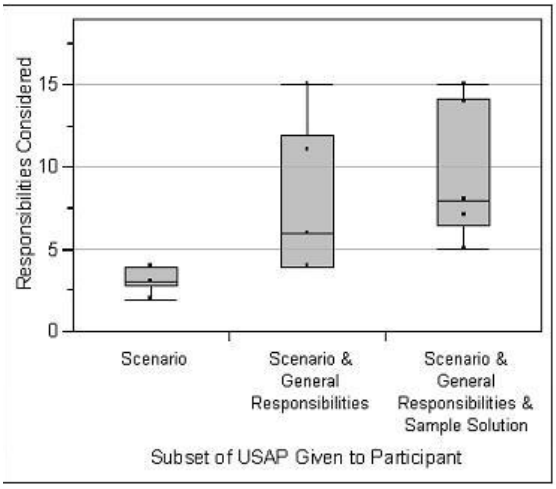

Figure 1: Results of laboratory experiment

better architecture design than those created by participants provided with just the scenario (p less than 0.05), but that the UML diagrams did not significantly improve the architecture design over the responsibilities alone. These results were reported in more detail in [2]. Figure 1 shows the results of the laboratory experiment.

Note, however, that there were 19 responsibilities in the

problem given to the participants in the laboratory study. Figure 1 shows that the group with the best performance achieved an average of only 9.5 responsibilities considered. That is, the participants' solutions, on average, only addressed half of the responsibilities that might have been considered.

D.4 Stakeholder choice of scenarios

The initial interactions between the ABB project team and the CMU research team consisted of information exchange about the project being developed and about the USAP approach. The researchers then presented 19 usability scenarios possibly relevant to this domain.

- Progress feedback

- Warning/status/alert feedback
- Undo
- Canceling commands
- User profile
- Help
- Command aggregation
- Action for multiple objects
- Workflow model
- Different views of data
- Keyboard shortcuts
- Reuse of information
- Maintaining compatibility with other systems
- Navigating within a single view
- Recovering from failure
- Identity management
- Comprehensive search
- Supporting internationalization
- Working at the user's pace

The ABB project team was asked to prioritize the general usability scenarios and they decided to focus on two and add an additional one. The chosen scenarios were User Profile and Alarms and Events (renamed from Warning/status/alert feedback). The additional scenario was Environment Configuration.

D.5 USAP Patterns

In the process of developing the three USAPs that were tested by the architects, we developed a Pattern Language [12], consisting of foundational USAPs and end-user USAPs, to exploit the commonalities among the USAPs. The pattern language was not visible to the architect and we will not describe it in this paper. The interested reader is referred to [13] for a description of the pattern language.

There are two aspects of the patterns on which we will focus. First there is an enumeration of textual responsibilities. These responsibilities are implementation independent. Collectively they cover the responsibilities necessary for implementing the three USAPs. There were 31 responsibilities for the architect to examine; 26 are shared by all three USAPs and 5 are specific to Alarms and Events. Each of the shared responsibility could pertain to each USAP and so the architect must consider 83 distinct situations.

An example of a responsibility is "The system must provide a means for an authorized author to save and/or export the [User Profile, Configuration description, Conditions for Alarms, Events and Alerts] (e.g., by auto-save or by author request). If other systems are going to use the [User Profile, Configuration description, Conditions for Alarms, Events and Alerts], then use a format that can be used by the other systems."

The portion of the responsibility that shows the three USAPs under consideration "[User Profile, Configuration description, Conditions for Alarms, Events and Alerts]" is an artifact that results from the Pattern Language. For each responsibility, we also provided implementation details. In the original formulation of USAPs, we provided UML patterns. This provision of UML followed the standard pattern writing advice of being very specific with respect to the patterns described. Three things made us replace the diagrams with "implementation details"

1. The results of the controlled experiment did not show a significant improvement in the participants that had access to diagrams over the participants that did not have access to diagrams.

2. Several ABB architects (not those involved in the product line development described here) felt that the diagrams were too judgmental. Since the diagrams in the solution were different than the diagrams of their architecture, they felt that they were being told they had designed their architecture incorrectly.

3. These architects also questioned whether it would be possible to integrate three (or more) different USAPs within the existing architecture. They had three different UML sample solutions and could not readily figure out how they should be integrated in practice.

The implementation detail provided for the responsibility quoted above is:

If the initiation of the save was automatic:
That portion of the system that manages the authoring process performs the initiation.
That portion of the system that manages the authoring process stores and/or exports the specification.

If the initiation of the save was at the author's request:
The portion of the system that renders output must render a UI that allows the parameters needed by the system (e.g., format, location) to be input and display them. The portion of the system that accepts input from the user must accept the parameters. That portion of the system that manages the authoring process stores and/or exports the specification.

Note that this is basically a textual description of what would be represented in a diagram. The structural elements of the implementation details are represented as "portions of the system" and the behavioral elements as activities performed by those portions of the system. By using the word "portion of the system" instead of a visual description in the form of a UML pattern, the designer can project the words onto her/his design and verify that the portion exists or, if not, design a new part in the solution corresponding to the "portion of the system" and its described activities. We will discuss the designers' reaction to the implementation guidance in the section on reactions.

D.6 Delivery tool

The challenge of encouraging the designers to consider all responsibilities was met by transferring the USAPs into a web-based tool [14]. The goals of the tool were ease-of-use, ease-of-understanding, helping the designers to actively consider all responsibilities, and the most important goal: bridging the gap between usability requirements from a set of general usability scenarios to software architecture requirements in the form of responsibilities.

The ease-of-use and ease-of-understanding goals are reflected in the tool by hiding the pattern language concepts of foundational USAPs and end-user USAPs from the user. The USAPs concept is instead visualized as a presentation of the foundational responsibilities hierarchy in the navigational menu without using the words “Foundational” or “End-User” (see Figure 2). In the main window each foundational USAP’s responsibilities are displayed with a pattern language parameter furnished by the prioritized end-user USAPs: Alarm & Events, User Profile, and Environment Configuration. Each responsibility has a check-box that is not checked by the architect, but by an internal state that is only is set to “check” when the designer has changed the state of the radio-button associated with each end-user USAP related to the responsibility. The radio-buttons states are set by the designer and reflects hers/his architecture’s state in relation to the responsibility and these are: “Architecture addresses this”, “Must modify architecture” and “Not applicable”. The state “Not yet considered” is the default state set when the designer has not yet made an active choice. The user can only make an adequate choice after reading the responsibility text thoroughly. Otherwise it would be difficult for the user to know her/his design’s state in relation to the responsibility. The entire layout of the USAP delivery tool was consciously made simple and direct. Additional informational text was hidden and displayed only when the user choose to display it by clicking a link, e.g. “Show rationale” for a responsibility. The help text could be hidden again by clicking a link, e.g. “Hide rationale.” We felt that the information content otherwise would be overwhelming for the users. The main page contained instructions on what a USAP is and how to use the USAP delivery tool. The states of the radio-buttons and check-boxes are persistent as long as the web-tool is open, enabling the user to go back and forth in the tool without losing data. Since the delivery tool was a prototype we did not take it to the level of a full-fledged content management tool with a database as the backbone. We wanted user feedback from the tests to inform the design before investing in this more expensive development step.

Figure 2 shows a screen shot of some of the responsibilities. If the designer wishes to discuss the responsibility with the remainder of the design team or other stakeholders, a check-box “Discuss this” can be checked by the designer. A future extension would be to add the possibility of including a comment for each responsibility. The interface of the tool encourages the designer to set the state of hers/his architecture in relation to each responsibility. The check-boxes next to each responsibility indicates to the designer whether the responsibility is fully considered for each USAP or not. These features are intended to address the problem that appeared in the laboratory studies of subjects not

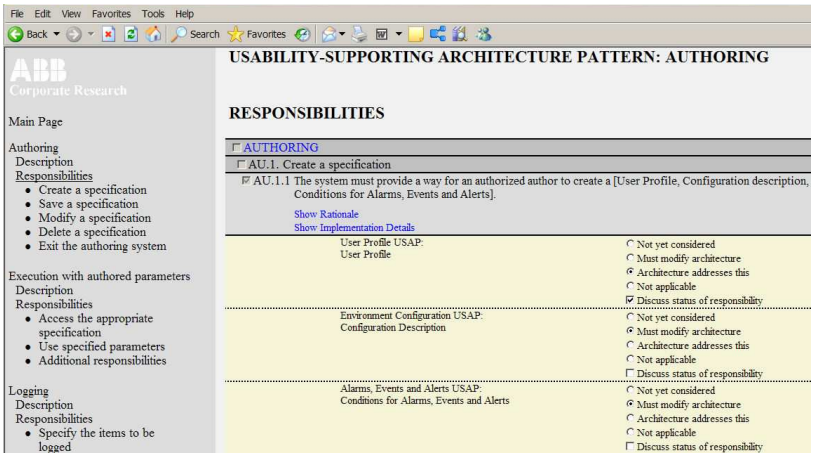

Figure 2: Prototype of a web-based interface for delivering USAP responsibilities to industry software architects

responding to half of the responsibilities.

It is also worth noting that the name of each of the three USAPs chosen for delivery is enumerated under the responsibility, and that the designer must respond to each responsibility in the context of each USAP. It is possible that state of the architecture will vary among the USAPs. Making the state of the architecture explicit with respect to each of the different USAPs will encourage the designer to consider each responsibility's applicability for each USAP. Presenting the three instances of each responsibility together, instead of organizing them by their USAP, encourages the architect to consider common design solutions.

Finally, observe that under each responsibility is a link that when clicked displays the implementation details as discussed above. When we discuss the results of using this tool, we will discuss how the designers made use of this feature.

Once the designers have considered and responded to all of the responsibilities, they can generate a "to do" list. This is a list of the responsibilities that either have not yet been considered or that require a modification of the architecture. Figure 3 shows a screen shot of the "to do" list generated by the screen shot in Figure 2. The "to do" list can then be incorporated into whatever project management scheme the designers use.

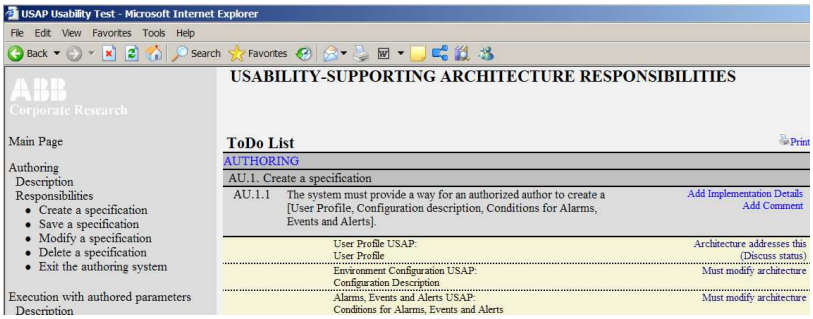

Figure 3: Prototype “to do” list produced from those responsibilities that are marked as requiring architectural modification

D.7 Results of using the USAP delivery tool

The two software architects from the product line system project used the USAP delivery tool at a time when they had completed a preliminary architecture design. One architect was senior and had created most of the preliminary design. The second architect had recently joined the project but had a solid background as software architect at an automobile company.

The Authorization foundational USAP was omitted from the test we performed in order to make the number of responsibilities tractable for a single day of testing. Since for the product line under development, authorization would not be needed, this did not impact the utility of the test from the point of view of evaluating the current design for support of the three chosen usability scenarios.

The two architects from the product line system project used the USAP delivery tool in one session lasting six hours interrupted by a one hour break for lunch and two 15-minute breaks for coffee. They examined and discussed each responsibility in turn, made notes as appropriate, and decided what response to make to that responsibility. In the six hours of work they completed consideration of all of the responsibilities for each of the USAPs. They averaged about 12 minutes per responsibility.

Overall the designers felt that the USAP delivery tool was quite helpful. Some of the quotes regarding the helpfulness of the tool:

Designer 1: *Yeah, I, I think it's, it's a very easy way to get some kind of review of your work. You will not get the complete picture of all your work, but it will be a very good check, or at least an indication of the completeness of your system.*

The main goal for ABB when applying the USAP technique was to incorporate usability support early in the design process in order to build in the support in the core architecture. By building in usability support early in the architecture, ABB expects to avoid late and costly redesign after the users have tested an actual version of the product line systems products. Some of the quotes that related to the goal of early architectural usability support were:

Designer 1: *We have discussed lots of internal stuff in the system but this gave us some picture of what the user is going to see.*

Designer2: *And that is things that we were not going to get that input, until very late in the design process, if we hadn't used this tool now. So it was good to have these points of view come in this early. I think we have identified at least a couple of new subsystems.*

Designer1: *Yes. And some shortcomings of the previous design.*

Designer2: *Yeah.*

The designers also responded well about the level of abstraction of the responsibilities:

Designer2: *The tool raises very abstract discussions and thoughts. It is much work to go through these responsibilities.*

Designer 2: *The most useful thing with this tool is that it guides your thoughts, and it helps you to think about the architecture that you have from different perspectives.*

From preliminary reactions at another ABB business where we showed the USAPs before removing the UML example and developing the pattern language, we were concerned that the designers would feel "supervised" or that they would feel that they had received unwanted and/or unhelpful recommendations. Instead, the reactions were very positive:

Designer 1: *It was like having a partner to discuss with.*
Designer 2: *The issues that you list in your tool, when you are sitting several people talking together about them, then you have to discuss how we handle these issues in our system, in our architecture. And that, that provides an understanding for the peoples who are important in the discussion, of how the architecture works.*

In contrast to the earlier negative reactions to UML diagrams of a sample solution, we found that as the designers examined the lists of responsibilities, they nearly always examined and discussed the implementation suggestions. One of their suggestions for improvement of the tool was that the implementation suggestions could be automatically included in the to-do list so that they would be available for future use, indicating that they saw these suggestions as useful instead of intrusive.

In summary, the reactions of the software architects to the tool were very positive. The designers had viewed all implementation details in a top-down fashion indicating that for every responsibility they felt it helpful to view the implementation guidelines. They also asked for a copy of the tool so that they could have it available as they worked through their to-do list.

During their use of the tool, the architects identified 14 issues that needed further consideration. Over the next several weeks, the architects considered these fourteen issues and their actual impact. The architects' judgment as to the resolution of each of the issues is detailed below.

Issue 1. Cost Saving:	- would have been done any way
Issue 2. Cost Saving:	- 1weeks
Issue 3. Cost Saving:	- weeks
Issue 4. Cost Saving:	- would have been done any way
Issue 5. Cost Saving:	- very uncertain of value
Issue 6. Cost Saving:	- very uncertain of value
Issue 7. Cost Saving:	- very uncertain of value
Issue 8. Cost Saving:	- 1 weeks
Issue 9. Cost Saving:	- very uncertain of value
Issue 10. Cost Saving:	- would have been done any way.
Issue 11. Cost Saving:	- very uncertain of value
Issue 12. Cost Saving:	- 2 weeks, could be more if this idea is fully exploited
Issue 13. Cost Saving:	- very uncertain of value
Issue 14. Cost Saving:	- very uncertain of value

For the issues where the architect felt secure in providing a value, 5 weeks were saved. Note the uncertainty of the architect with respect to many of the other issues. In the worst case, this uncertainty translates to no additional savings but, likely, there were additional savings beyond that estimated initially. In any case, saving 25 days (5 weeks) for less than one day of investment by two people is still an amazing result.

The savings does not include the time the researchers have invested in producing the USAPs but Alarms and Events and user profiles are common usability scenarios. These USAPs are reusable across many projects and thus the investment to produce them will get amortized across multiple projects.

D.8 Conclusions and Future Work

On the one hand, providing professionals with a check list of activities they should perform is a very old concept. Computerizing the checklist is not a major step. The resulting tool is extremely simple. On the other hand, getting a 25-to-2 return on investment (ROI) for the architects - one day's work by two people saved five weeks - is an amazing result. One study with one estimate is not scientific evidence but this study is one of the few reports of ROI with respect to the use of any architectural technique. Architectural knowledge can be encoded into very simple tools and still be effective. Architectural tool builders might consider simple methods to encode their knowledge rather than

attempting very sophisticated tools. Furthermore, three aspects of this work are significant.

1. The patterns are primarily described at the level of responsibilities. These are independent of implementation, and lead the architects to think about how a particular responsibility relates to their current system design rather than forcing them to attempt to compose structural instructions with their current design.

2. Using textual descriptions for implementation instructions rather than diagrams was well received by the architects at ABB. The push back from architects with respect to diagrammatic instructions has not previously been reported.

3. Encouraging the architects through a tool to examine all of the items in the checklist removes the problems with paper delivery of the checklist.

In addition, there is nothing in the USAP delivery tool that is specific to usability patterns. Any quality attribute where the requirements can be expressed as a set of responsibilities, e.g. security, could likely be included in the tool. The same portions of a system could then be represented in both a security responsibilities implementation details and in a usability responsibilities implementation details.

D.9 Acknowledgments

The Software Engineering Institute is a Federally Funded Research and Development Center created by the US Department of Defense. A portion of the third author's time on this research was funded by the Institute of Education Sciences, US Department of Education, through Grant R305B040063 to Carnegie Mellon University, and by ABB. The views and conclusions herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of IES, SEI, the U.S. Government, or ABB.

Bibliography

[1] R. J. Adams, L. Bass, and B. E. John. *Applying general usability scenarios to the design of the software architecture of a collaborative workspace.* In A. Seffah, J. Gulliksen and M. Desmarais (Eds.) Human-Centered Software Engineering: Frameworks for HCI/HCD and Software Engineering Integration. Kluwer Academic Publishers, 2005.

[2] E. Golden, B. E. John, and L. Bass. The value of a usability-supporting architectural pattern in software architecture design: A controlled experiment. In *Proceedings of the 27th International Conference on Software Engineering, ICSE*, St. Louis, Missouri, May 2005.

[3] B. E. John, L. Bass, M.-I. Sanchez-Segura, and R.J. Adams. Bringing usability concerns to the design of software architecture. In *Proceedings of The 9th IFIP Working Conference on Engineering for Human-Computer Interaction and the 11th International Workshop on Design, Specification and Verification of Interactive Systems*, Hamburg, Germany, 2004.

[4] E. Folmer. *Software Architecture Analysis of Usability*. PhD thesis, Department of Computer Science, University of Groningen, Groningen., 2005.

[5] E. Folmer, J. van Gurp, and J. Bosch. A Framework for capturing the Relationship between Usability and Software Architecture. Software Process: Improvement and Practice, Volume 8, Issue 2. Pages 67-87., 2003.

[6] N. Juristo, A.M. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *Software Engineering, IEEE Transactions on*, 33(11):744–758, Nov. 2007.

[7] M. Barbacci, R. Ellison, A. Lattance, J. Stafford, C. WeinStock, and W. Wood. Quality attribute workshops, 3rd edition. Technical report, Software Engineering Institute, Pittsburgh, PA, USA, 2003.

[8] P. Stoll, A. Wall, and C. Norström. Guiding Architectural Decisions with the Influencing Factors Method. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008, 2008.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture A System of Patterns*, volume 1. Wiley, first edition, 1996.

[11] J. Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media: Sebastopol, CA, 2006.

[12] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA, 1977.

[13] B. E. John, L. Bass, E. Golden, and P. Stoll. A responsibility-based pattern language for usability-supporting architectural patterns. Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS), Pittsburgh, PA, US, 2009.

[14] P. Stoll, L. Bass, B. E. John, and E. Golden. Preparing Usability Supporting Architectural Patterns for Industrial Use. Proceedings of International Workshop on the Interplay between Usability Evaluation and Software Development (I-ISED), Pisa, Italy, 2008.

Appendix E

Paper E: Software Engineering featuring the Zachman Taxonomy

Pia Stoll, Anders Wall
Industrial Software Systems
ABB Corporate research
pia.stoll@se.abb.com,
anders.wall@se.abb.com

Christer Norström
Computer Science and Electronics
Mälardalen University
christer.norstrom@mdh.se

Technical Report, School of Innovation, Design and Engineering (IDT), Mälardalen University, Sweden, 2009.

Abstract

Sustainable development of industrial software system companies is related to aspects of the architecture's environment with influences from organization, business and architecture. The definition of sustainable development states that a development organization must meet the needs of the organization's stakeholders, without compromising its ability to meet the needs of future stakeholders as well. Classifying the software development artifacts in a framework of views, from the macro-design level down to the micro-design level, would give a deeper understanding of the forces that act between the sustainable software development artifacts.

In this report, the Software Engineering Taxonomy is presented which is derived from the Zachman Enterprise Architecture Framework. Based on two assumptions, the Software Engineering Taxonomy proved to be able to classify all software engineering artifacts from the IEEE Software Engineering Book Of Knowledge (SWEBOK) published 2004.

The Software Engineering Taxonomy also proved to give useful insights into how customer sites and development sites may interact for fast innovation exemplified with the companies Apple (AppStore) and Google. The taxonomy also proved to be useful for process analysis which is shown for the Scrum process.

E.1 Introduction

This report investigates the possibility of classifying software engineering artifacts for industrial software systems. The classification should include artifacts related to business and organization and therefore three Enterprise Architecture frameworks were considered. The three frameworks were: the Zachman framework, the Department Of Defense Architecture Framework (DODAF) [1] and The Open Group Architecture Framework (TOGAF) [2].

The discipline of enterprise architecture is commonly considered to have its birth in an academic article by Zachman, published 1987 by the research oriented IBM Systems Journal [3]. Zachman saw the growing complexity of information software system that extended in scope and complexity to cover an entire enterprise. He stated that decentralization of system resources without structure results in chaos and argued for the need of information system architecture. Zachman searched for an objective independent basis upon which to build a framework for information system architecture and resolved to be inspired by classic architecture. After some enhancements [4], the result was a classification framework, a taxonomy, of 30 cells representing the intersections of usage perspectives and content abstractions of architectural information.

Enterprise[1] architecture as defined by the Federal Architecture Working Group (FAWG) [5] is: a strategic information asset base and describes the mission (i.e. the business), the information and the technologies necessary to perform the mission, and the transitional processes for implementing new technologies in response to changing mission needs. An enterprise architecture includes a baseline architecture[2], target architecture[3], and a sequencing plan[4].

According to James N. Martin [6] enterprise architecture deals with "Getting to the Future" and has drivers and outcomes. The enterprise architecture is according to Martin a means for transforming enterprise objectives into business plans and mission needs.

In the mid 1990s the DOD determined that a common approach was needed for describing its architectures, so that DOD systems could efficiently communicate and inter-operate during joint and multinational operations. The interop-

[1] Enterprise - an organization supporting a defined business scope and mission. An enterprise includes interdependent resources (people, organizations, and technology) who must coordinate their functions and share information in support of a common mission.

[2] Baseline architecture - the architecture as it is today, also called as-is architecture

[3] Target architecture - the (planned) future architecture, also called to-be architecture or goal architecture

[4] Sequencing plan - the strategy for changing the baseline architecture to the target architecture, also called the transition plan

erability aspects of the DODAF is reflected in its architectural views which are focused on describing what's being communicated and how in the Operational View (OV) of the DODAF. The Systems View (SV) of DODAF identifies the systems that support the OVs and the Technical View (TV) describes the criteria for each required system that will satisfy the interoperability requirements. DODAF is as such not an architecture development method or a classification framework, it's an architecture description development framework focused on describing interoperability aspects of systems of systems.

TOGAF[5] is developed and maintained by members of The Open Group, working within the Architecture Forum. The original development of TOGAF Version 1, in 1995, was based on the Technical Architecture Framework for Information Management (TAFIM), developed by the US Department of Defense (DOD). The DOD gave The Open Group explicit permission and encouragement to create TOGAF by building on the TAFIM, which itself was the result of many years of development effort and many millions of dollars of US Government investment.

TOGAF is more ambitious in scope than its defense counterpart, DODAF. TOGAF organizes architectures into four domain levels:

Business architecture - defines business strategy, governance, organization, and key business processes

Application architecture - specifies individual application systems to be deployed

Data architecture - defines structure of an organization's logical and physical data assets and associated data management resources

Technology architecture - specifies software infrastructure intended to support the deployment of core, mission-critical applications

As this report was searching for a enterprise architecture artifact classification framework, not an enterprise architecture description development framework or in-house information system architecture development framework, it resorted to study the Zachman framework in more detail.

The remainder of this report is organized as follows; Section E.2 describes the Zachman Framework, Section E.3.1 describes the Software Engineering Taxonomy and the classification of the SWEBOK software engineering artifacts, Section E.3.3 and Section E.3.4 uses the Software Engineering Taxonomy from Section E.3.1 to analyze the cases: AppStore, Google and Scrum,

[5] http://www.opengroup.org/architecture/togaf9-doc/arch/ [Accessed: 4. February 2009]

and Section E.4 presents the conclusions of the work with the Software Engineering Taxonomy and its usefulness for the software engineering discipline and future work.

E.2 Zachman Framework

In a joint article [4], published 1992, Sowa and Zachman explain that the Zachman framework links the concrete things in the world (entities, processes, locations, people, times and purposes) to the abstract bits in the computer. The Zachman framework is not a replacement of programming tools, techniques, or methodologies but instead, it provides a way of viewing the system from many different perspectives and how they are all related. The framework logic can be used for describing virtually anything considering its history of development. The logic was initially perceived by observing the design and construction of buildings. Later it was validated by observing the engineering and manufacture of airplanes. Subsequently, it was applied to enterprises during which the initial material on the framework was published [3][7][8]. Sowa and Zachman write:

> *Most programming tools and techniques focus on one aspect or a few related aspects of a system. The details of the aspect they select are shown in utmost clarity, but other details may be obscured or forgotten. By concentrating on one aspect, each technique loses sight of the overall information system and how it relates to the enterprise and its surrounding environment. The purpose of the ISA framework [Today, the Zachman framework A.R.]is to show how everything fits together. It is a taxonomy with 30 boxes or cells organized into six columns and five rows. Instead of replacing other techniques, it shows how they fit in the overall scheme.*

According to Zachman, “Architecture” is the set of descriptive representations relevant for describing a complex object (actually, any object) such that the instance of the object can be created and such that the descriptive representations serve as the baseline for changing an object instance.

The columns of the framework represent different abstractions from or different ways to describe information of the complex object. The reason for isolating one variable (abstraction) while suppressing all others is to contain the complexity of the design problem. Abstractions classifying the description focus are:

Abstraction / Perspective	INVENTORY SETS (WHAT)	PROCESS TRANSFORMATIONS (HOW)	NETWORK NODES (WHERE)	ORGANIZATION GROUPS (WHO)	TIMING PERIODS (WHEN)	MOTIVATION REASONS (WHY)
SCOPE CONTEXTS (Strategists)	e.g. Inventory Types	e.g. Process Types	e.g. Network Types	e.g. Organization Types	e.g. Timing Types	e.g. Motivation Types
BUSINESS CONCEPTS (Executive Leaders)	e.g. Business Entities & Relationships	e.g. Business Transform & Input	e.g. Business Locations & Connections	e.g. Business Role & Work	e.g. Business Cycle & Moment	e.g. Business End & Means
SYSTEM LOGIC (Architects)	e.g. System Entities & Relationships	e.g. System Transform & Input	e.g. System Locations & Connections	e.g. System Role & Work	e.g. System Cycle & Moment	e.g. System End & Means
TECHNOLOGY PHYSICS (Engineers)	e.g. Technology Entities & Relationships	e.g. Technology Transform & Input	e.g. Technology Locations & Connections	e.g. Technology Role & Work	e.g. Technology Cycle & Moment	e.g. Technology End & Means
COMPONENT ASSEMBLIES (Technicians)	e.g. Component Entities & Relationships	e.g. Component Transform & Input	e.g. Component Locations & Connections	e.g. Component Role & Work	e.g. Component Cycle & Moment	e.g. Component End & Means

Figure 1: The Zachman Framework

Inventory Sets - Describes ‘what” information is used

Process Transformations - Describes “How” the information is used

Network Nodes - Describes “Where” the information is used

Organization Groups - Describes “Who” is using the information

Timing Periods - Describes “When” the information is used

Motivation Reasons - Describes “Why” the information is used

The rows of the framework represent “Perspectives” classifying the description usage. The perspectives are:

Scope Contexts - perspective descriptions corresponds to an executive summary for a planner or investor who wants an estimate of the scope of the system, what it would cost, and how it would perform.

Business Concepts - perspective is the perspective of the owner, who will have to live with the constructed object (system) in the daily routines of business. This perspective descriptions correspond to the enterprise (business) model, which constitutes the design of the business and shows the business entities and processes and how they interact.

System Logic - perspective is the designer's perspective. The System Logic perspective descriptions correspond to the system model designed by a systems analyst who must determine the data elements and functions that represent business entities and processes.

Technology Physics - perspective descriptions correspond to the technology model, which must adapt the system model to the details of the programming languages, I/O devices, or other technology. This is the perspective where the four views of the "4+1" model by Kruchten [9] can be used to describe software architecture.

Component Assemblies - perspective descriptions correspond to the detailed specifications that are given to programmers who code individual modules without being concerned with the overall context or structure of the system.

The relevant descriptive representations would necessarily have to include all the intersections between the Abstractions and the Perspectives (Figure. 1). "Architecture" would be the total set of descriptive representations (models) relevant for describing the complex object and required to serve as a baseline for changing the complex object once it is described. Zachman's complex object is the enterprise, but principally he states that the complex object can be any object.

The Zachman framework is a structure, not a methodology for creating the implementation of the object. The Zachman Framework does not imply anything about how architecture is done (top-down, bottom-up, etc). The level of detail is a function of a cell not a function of a column. The level of detail needed to describe the Technology Physics perspective may be naturally high but it does not imply that the level of detail of the Scope Contexts descriptions should be lower or the opposite.

The framework is normalized, that is adding another row or column to the framework would introduce redundancies or discontinuities. Composite models and process composites are needed for implementation. A composite model is a model that is comprised of elements from more than one framework

model. For architected implementations, composite models must be created from primitive models and diagonal composites from horizontally and vertically integrated primitives. The structural reason for excluding diagonal relationships is that the cellular relationships are transitive. Changing a model may impact the model above and below in the same column and any model in the same row.

The rules of the framework are [7]:

Rule 1: Do not add rows or columns to the framework

Rule 2: Each column has a simple generic model

Rule 3: Each cell model specializes its column's generic model

Rule 3 Corollary: Level of detail is a function of a cell, not a column

Rule 4: No meta concept can be classified into more than one cell

Rule 5: Do not create diagonal relationships between cells

Rule 6: Do not change the names of the rows or columns

Rule 7: The logic is generic, recursive

The model, i.e. the view, in the Zachman framework can be aligned with the ISO/IEC 42010:2007 viewpoints [10]:

> *An organization desiring to produce an architecture framework for a particular domain can do so by specifying a set of viewpoints and making the selection of those viewpoints normative for any Architectural Description claiming conformance to the domain-specific architectural framework. It is hoped that existing architectural frameworks, such as the ISO Reference Model for Open Distributed Processing (RM-ODP) [11], the Enterprise Architecture Framework of Zachman [3], and the approach of Bass, Clements, and Kazman [12] can be aligned with the standard in this manner.*

Zachman's framework does not describe what language to use for the model descriptions or how to do the actual modeling for each cell. Therefore each view of the Zachman's framework is free to use the viewpoint selected by the responsible of the description. It should therefore be possible to use the viewpoints from the ISO/IEC 42010:2007 to describe a model, i.e. a view, within the framework.

For manufacturing a process composite would be necessary. The process composite describes the working process of creating the model descriptions of the composite model, typically ending with the descriptions of the components in the Component Assemblies perspective, e.g. a service or framework. A third dimension of the framework, called science, has been proposed by O'Rourke et al. [13]. This extension is known as the Zachman DNA (Depth iNtegrating Architecture). In addition to the perspectives and aspects the z-axis is used for classifying the practices and activities used for producing all the cell representations.

E.3 Software Engineering Taxonomy

In order to be able to use the Zachman framework for software engineering artifacts, two basic assumptions were done:

1. The software engineering classification framework, derived from the Zachman framework, describes the software system's development organization and the customer's scope and business related to the need of system support.
2. The software engineering classification framework, derived from the Zachman framework, is three-dimensional where site is the third dimension. The site might be the software development organization, external development organization or the customer's enterprise as long as the site has a part in the system usage or system development.

The assumptions are illustrated in Figure 2. With these assumptions, the system development's Business Concepts perspective will describe the software development artifacts, e.g. software development activities, software development team locations and connections, software development roles and work products, software development schedules, and software development strategies. The models in the customer's Business Concepts perspective will describe the customer's production related to the need of system support. The resulting software engineering classification framework is called the Software Engineering Taxonomy.

E.3.1 Shared Perspectives

The models in the Software Engineering Taxonomy might be shared across development and customer sites but it is the software development organization

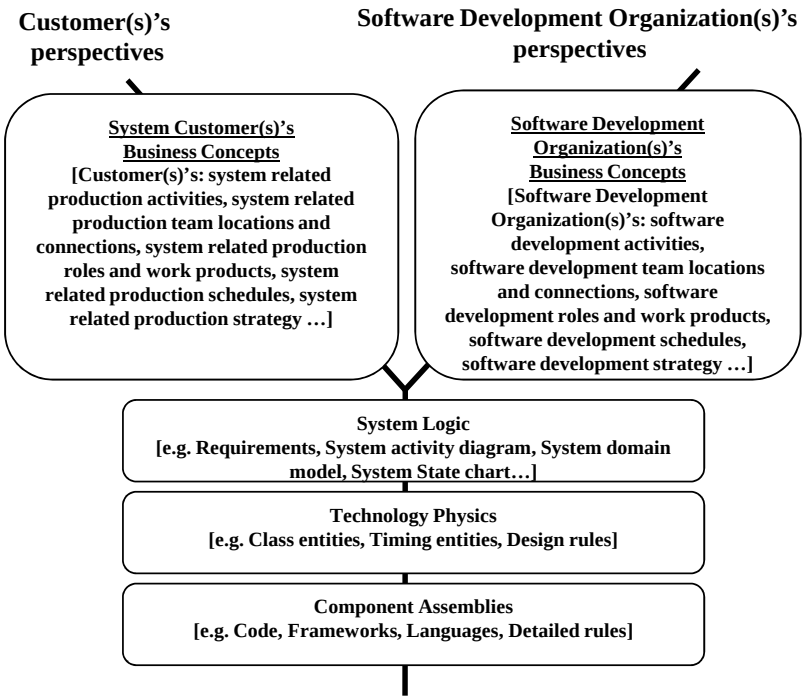

Figure 2: The Customer's and the Software Development Organization's perspectives

that controls the degree of openness. For example if the customer is an active member in the requirements handling team at the software development organization, then the requirements handling activity is shared across sites. This would mean that the model with the Business Concepts perspective and Process Transformations abstraction is partly shared since this model contains the requirements handling activity. Another example of shared models across sites is the open source development. In open source development, several software development sites share software development model descriptions across sites. Not only the development activities can be shared across sites, but also the testing activities. Google lets their customers test the Google software applications before the final release, which makes the customers part of the test team.

E.3.2 Software Engineering Descriptions

The IEEE Software Engineering Body Of Knowledge, SWEBOK, has the objective to promote a consistent view of software engineering worldwide and was published 2004 [14]. SWEBOK has references to a very large number of software engineering theories. The SWEBOK has divided the software engineering domain into a set of knowledge areas; software requirements, software design, software construction, software testing, software maintenance, software configuration management, software engineering management, software engineering process, software engineering tools and methods, and software quality. The knowledge areas act as knowledge for the persons working in that specific area. In contrast to the work described in [15][16] the classification of the SWEBOK software engineering artifacts in this report does not try to reflect the software engineering professions but instead seek a classification approach similar to the building engineering [17] .

The process used to classify the descriptions from the SWEBOK was:

1. Anything resembling an artifact was extracted from the SWEBOK.
2. The artifact duplicates were removed when all of the artifacts were extracted.
3. The non-duplicate artifacts were analyzed and grouped according to their descriptions. For example, the “Release Schedule” artifact was grouped together with “Construction Schedule”, “Project Schedule and milestones”, and “Test Schedule”.
4. The grouped artifacts were translated into general model descriptions. For example, the group of schedules in the previous step was generalized into the model description “Schedules for projects, releases and processes”.
5. The general model descriptions were classified according to the perspectives and abstractions from the Software Engineering Taxonomy.

Abstraction → / Software Development Organization Perspective ↓	INVENTORY SETS (WHAT)	PROCESS TRANSFORMATIONS (HOW)	NETWORK NODES (WHERE)
SCOPE CONTEXTS	[Reports, Standards, Stakeholders, Tools, Products, Programming Languages, Developer Competencies]	[Requirement Handling/ Design/ Construction/ Testing/ Maintenance/ Configuration Management/ Engineering Management]	[Internal & External Development Team Networks, Supplier Networks]
BUSINESS CONCEPTS	[Estimations, Prototypes, Analysis, …, Decisions and their Relations]	[Activities for: Requirement Handling/ Design/ Construction/ Testing/ Maintenance/ Configuration Management/ Engineering Management]	[Internal & External Development Team Locations & Connections]
SYSTEM LOGIC	[System Domain Model]	[System Activity Diagram]	[System Deployment Diagram]
TECHNOLOGY PHYSICS	[Development View, Class Diagram]	[Logical View, Interface Specification]	[Physical View, Deployment Diagram]
COMPONENT ASSEMBLIES	[Database Configuration, Build Configuration]	[Algorithms, Code Modules, Frameworks]	[Communication Protocols, Port Configurations]

Figure 3: The SWEBOK software engineering artifacts classified in the Software Engineering Taxonomy. The figure shows the three first columns.

The software engineering artifacts are not physical like in the building engineering but differ in their descriptions, not in their physical dimensions. A software engineering description can be very complex, e.g. a domain model including a large set of entities and their relations, and it can be less complex, e.g. the listing of reports. Some of the software engineering artifacts from SWEBOK could be descriptions of their own. For example, the artifact "System Class diagram" could be a complete description of the model with the Inventory Sets abstraction and the Technology Physics perspective. When performing the classification it was important to distinguish between the customer's perspec-

ORGANIZATION GROUPS (WHO)	TIMING PERIODS (WHEN)	MOTIVATION REASONS (WHY)	← Abstraction Software Development Organization ↓ Perspective
[External Regulatory Bodies, Internal & External Development Teams]	[Internal & External Development Releases, Global Economy Events, …]	[Business Goals, Process Scopes, Policies, Culture, Principles, Missions]	SCOPE CONTEXTS
[Internal & External Development Team Roles & their Work Products]	[Schedules for Projects/ Releases/ Processes]	[Strategies for: Processes'/ Projects'/ Staffing/ System and Projects' Objectives]	BUSINESS CONCEPTS
[System Use Cases]	[State Charts]	[Requirements, Constraints, Qualities]	SYSTEM LOGIC
[User Interfaces]	[Sequence Diagrams]	[Design Rules, Design Principles]	TECHNOLOGY PHYSICS
[Security Control, Safety Control]	[Concurrency Model]	[Explicit Design Rules/ Design Principles Configuration]	COMPONENT ASSEMBLIES

Figure 4: The SWEBOK software engineering artifacts classified in the Software Engineering Taxonomy. The figure shows the three last columns.

tives and the development organization's perspectives. For example, the model description "Schedules for projects, releases and processes" was classified with a Timing Periods abstraction and Software Development Organization's Business Concepts perspective. The software engineering artifacts which describe the development of a software system and the system itself were expected to be straight-forward to classify in Zachman's System Logic perspective since this perspective contains model descriptions of software system architecture, e.g. use cases, activity diagrams, requirements.

It was also expected that Zachman's Business Concepts perspective would

be difficult to use for Software Engineering artifacts since this perspective is typically used to model the customer's business processes in need of system support, e.g. production processes. The classification showed that the development organization's Business concepts as e.g. software testing, development schedules, prototype analysis etc could easily be classified in the Software Engineering Taxonomy when the taxonomy perspectives were the software development organization perspectives.

Non of the SWEBOK software engineering artifacts described the customer's Scope Contexts or customer's Business Concepts perspective and hence the classification of the SWEBOK artifacts is done using only the software development organization's perspectives. The resulting classification of the SWEBOK artifacts is two-dimensional and shown in Figure 3 and in Figure 4.

The generalized model descriptions are enclosed by brackets in each cell of the Software Engineering Taxonomy in Figure 3 and Figure 4. For example, the cell with the Inventory Sets abstraction and Scope Contexts perspective contains descriptions of reports, stakeholders, standards, tools and products used by the software development organization. The cell with the Process Transformations abstraction and Scope Contexts perspective contains model descriptions of processes for requirement handling, design, construction, testing, maintenance, configuration management, and engineering management. In the Software Engineering Taxonomy, the processes do not dictate the classification; they are a part of the classification scheme. The Scope Contexts- and Business Concepts perspectives with the Process Transformations abstraction got a large number of artifacts classified since SWEBOK contains a large amount of process descriptions and activity definitions for the processes. The models description would be instantiated for each software development organization. For example, an organization doing Scrum [18] processes would instantiate the model with the Business Concepts perspective and Process Transformations abstraction with descriptions of typical Scrum activities: "Sprint Review", "Planning", etc.

E.3.3 Apple and Google Process Composite Models

The interactions between development sites and customer/utilization sites are re-engineered into the Software Engineering Taxonomy for two companies' applications: Apple's AppStore [[19], [20], [21]] and Google's services [22]. The companies are world-leading [23] in establishing new ways of interacting with their customers during software development and therefore highly interesting for creating composite models which bridge the gap between cus-

tomer/utilization site(s) and development organization site(s) in the Software Engineering Taxonomy.

Apple has created a way to easily install applications in run-time by structuring application code into bundles [21]. The bundle structure is part of the Apple framework. Apple shares the framework but in contrast to the open source community gives external developers no access to the Apple core business logic components.

The shared composite model pattern for bridging the utilization- and development sites gap for Apple and Google is visualized in, Figure 5. The innovative integration takes place in the Network Nodes abstraction in the Software Engineering Taxonomy for both AppStore and Google services.

The AppStore describes the connections of internal- and external developers, customers, and the Apple organization through the Internet and through the mobile phone network.

The customers get a test/product strategy role when they indirectly drive both the internal and external development by downloading the internally and externally developed applications. The top-ten download list is visible for customers as well as developers on the AppStore web page.

Google's Ecosystem [22] describes the global locations and connections of Google's systems, services, advertisers, and customers over locations barriers world-wide. Google makes services available for external sites to use in their applications via standard protocols. Customers get a test/product strategy role when they test beta-versions of Google's products voluntarily.

The system design and deployment are crucial but not shared since they are descriptions of the core business logic. By considering what models in the Software Engineering Taxonomy are possible to share with external sites, new ways of bridging the gap between utilization and development can be found, which could create faster innovation of new or enhanced products.

E.3.4 Scrum Composite Process Model

When reverse-engineering the Scrum process, as described by Schwaber in [18], into our Software Engineering Taxonomy it becomes clear that the Scrum approach is rather extensive in the scope- and business perspective (Figure 6). To bridge the gap between customer/utilization site and developer site, the Scrum process includes the customer and the sales organization as members of the development team. By integrating customer, management, release management, and development in a set of teams, all the teams' concerns are integrated in a dynamic team work product called "product backlog". Two important

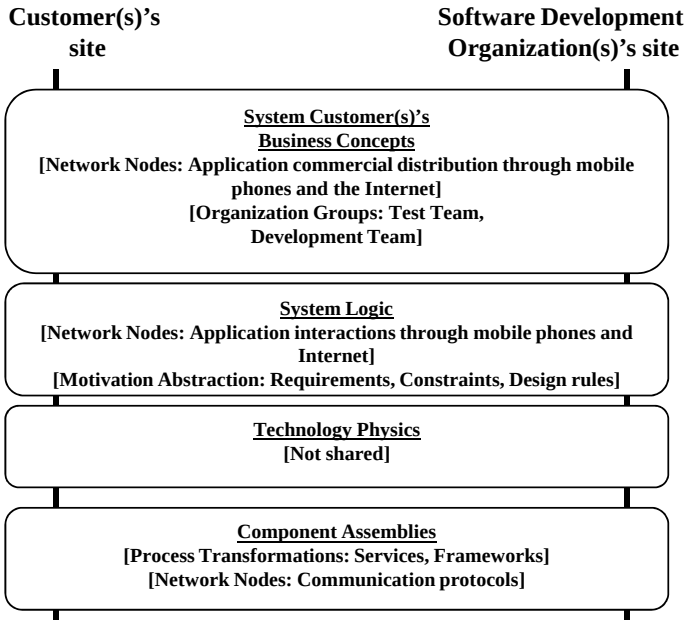

Figure 5: Bridges between Customer Site(s) and Development Site(s) for Apple and Google.

activities in the Scrum development process is the cost estimations and risk estimations.

The requirements and qualities are described in the Software Engineering Taxonomy cell with the Motivation Reasons abstraction and System Logic perspective. The code or program is described in the cell with the Process Transformations abstraction and Component Assemblies perspective. The composite process model, the Scrum development process as described in [18], takes a step from requirements to architectural design and domain modeling in the pregame phase. If the Scrum composite process model would have taken a direct step from requirements to code, then Zachman's rule no:5 stipulating "Do not create diagonal relationships between cells" would have been violated.

An interesting approach would be to integrate explicitly formulated design rules [24], described in the taxonomy cell with the Motivation Reasons abstraction and the Technology Physics perspective. This would be an alternative way

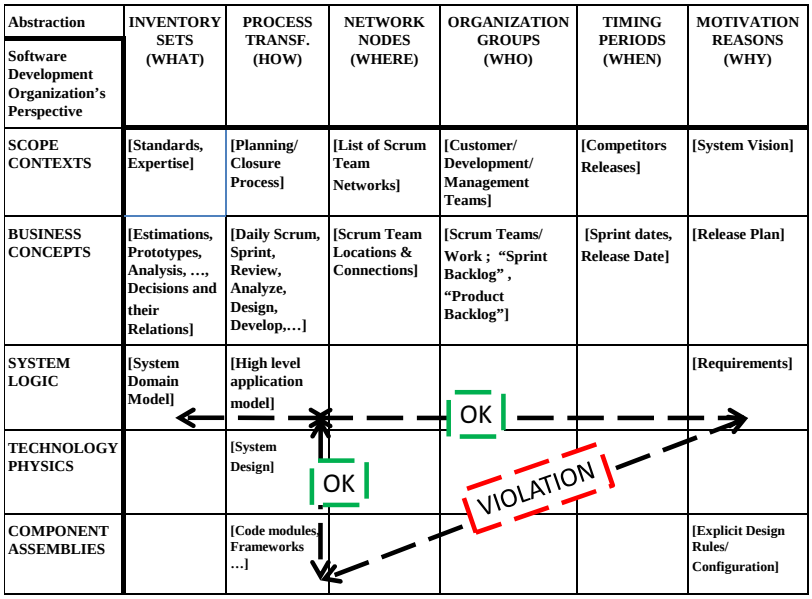

Figure 6: Scrum reverse engineered into the Software Engineering Taxonomy

or additional step to take from requirements to code.

E.4 Conclusions and Future Work

The Software Engineering Taxonomy derived out of the Zachman Framework relies on two assumptions:

1. The software engineering classification framework, derived from the Zachman framework, describes the software system's development organization and the customer's scope and business related to the need of system support.

2. The software engineering classification framework, derived from the Zachman framework, is three-dimensional where site is the third dimension. The site might be the software development organization, external development organization or the customer's enterprise as long as the site has a part in the system usage or system development.

The classification of the IEEE SWEBOK artifacts uses only the software development organization's perspectives, not the customer perspective, resulting in the classification being two-dimensional. However, the three dimensions of the Software Engineering Taxonomy can be used to describe a software development organization that shares models with external software development sites or customer sites, e.g. Google, Apple's AppStore and Open Source development as described in this report. The analysis of AppStore and Google showed that the taxonomy's Network Nodes abstraction and Organization Groups abstraction columns are important for sharing models with external development- and utilization sites for faster innovation of new products.

The reverse engineering of the Scrum process into our Software Engineering Taxonomy showed that all of the Scrum artifacts can be classified and that the focal point of the Scrum is on the Scope Contexts perspective and the Business Concepts perspective of the development organization. The descriptions of the System Logic perspective and the Technology Physics perspective are thin in the Scrum process.

The Software Engineering taxonomy can serve as a reasoning framework into which artifacts and results of software engineering theories, processes and case studies might be mapped for further analysis. The rules of the Zachman framework are valid for the Software Engineering Taxonomy.

It remains to do a formal validation of the Software Engineering Taxonomy. The formal validation could be in the form of a more thorough collection of software engineering artifact and their classification. Further, an expert panel could judge the classification's correctness.

Bibliography

[1] DoD. Department of Defence Architecture Framework Working Group, DoD Architecture Framework, DoDAF, version 1.0. Department of Defence, 2003.

[2] TOG. *The Open Group Architecture Framework, version 8/9, 2002/6*. The Open Group,.

[3] J. A. Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[4] J. F. Sowa and J. A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM System Journal*, 31:590–616, 1992.

[5] R. C. Thomas. A Practical Guide to Federal Enterprise Architecture,. www.gao.gov/bestpractices/bpeaguide.pdf, 2001. retrieved July 11th 2009.

[6] J. N. Martin. *An introduction to the Architectural Frameworks DODAF/MODAF/NAF*. Course given at the Royal Institute of Technology, Stockholm, Sweden, 2006.

[7] J. A. Zachman. *The Zachman Framework for Enterprise Architecture; A Primer for Enterprise Engineering and Manufacturing*. Zachman International, 2003.

[8] J. A. Zachman. The Zachman Framework and Observations on Methodologies. *Business Rules Journal*, 5(11), 2004.

[9] P. B. Kruchten. The “4+1” View Model of architecture. *Software, IEEE*, 12(6):42–50, Nov 1995.

[10] R. Hilliard. Systems and software engineering - Recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1–24, 15 2007.

[11] ISO/IEC 10746 - 3: 1996, Information technology - Open distributed processing - Reference model: Architecture, 1996.

[12] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, second edition, 2003.

[13] C. O'Rourke, N. Fishman, and W. Selkow. Enterprise Architecture, Using the Zachman Framework. *Thomson Course Technology*, 2003.

[14] P. Bourque and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[15] O. Mendes and A. Abran. Software Engineering Ontology: A Development Methodology. Technical report, University from Quebec in Montreal, 2004.

[16] P. Wongthongtham, E. Chang, and I. Sommerville. Software Engineering Ontology for Software Engineering Knowledge Management in Multi-site Software Development Environment. http://smi.stanford.edu/projects/protege/conference/2007/presentations, 2007.

[17] ASTM. ASTM Standard C33, "Specification for Concrete Aggregates", 2003.

[18] K. Schwaber. Scrum development process. Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. ACM/SIGPLAN OOPS Messenger 6(4), October 1995.

[19] D. B. Yoffie and M. Slind. Apple computer, 2006, 2007.

[20] P. Tsarchopoulos. Innovation lessons from apple. *The Economist*, 2007.

[21] Apple. About bundles, 2005.

[22] B. Iyer and T. H. Davenport. Reverse engineering google's innovation machine. *Harvard Business Review*, 2008.

[23] J. McGregor. The world's 50 most innovative companies. *Business Week*, 2008.

[24] M. J. LaMantia, Y. Cai, A. D. MacCormack, and J. Rusnak. Evolution analysis of large-scale software systems using design structure matrices and design rule theory. *Harvard Business School Working Knowledge*, 2007.

Appendix F

Paper F: Applying the Software Engineering Taxonomy

Pia Stoll, Anders Wall
Industrial Software Systems
ABB Corporate research
pia.stoll@se.abb.com,
anders.wall@se.abb.com

Christer Norström
Computer Science and Electronics
Mälardalen University
christer.norstrom@mdh.se

Technical Report, School of Innovation, Design and Engineering (IDT), Mälardalen University, Sweden, 2009.

Abstract

The Software Engineering Taxonomy is a derivative of the Zachman framework. Being a derivative of the Zachman framework, the Software Engineering Taxonomy follows the Zachman consistency rules and incorporates traditional enterprise architecture views together with software engineering views. In this report, the Software Engineering Taxonomy is applied as a reasoning framework in three studies: the Influencing Factors method field study, the Usability-Supporting Architecture Patterns field study, and the Sustainable Industrial Software Systems case study.

Software engineering artifacts from the three studies are extracted and classified in the Software Engineering Taxonomy. From the classification of data from the studies, it's shown that each one of the studies uses a subset of the thirty views in the Software Engineering Taxonomy to describe a specific method or theory. What views are used, depends on the scope of the researched object. In the classification of the USAP study artifacts, eight views were used in contrast to the Sustainable System study, that used nineteen views. This shows that the scope and interrelation complexity of sustainable development is much higher than the scope and interrelation complexity of the usability-supporting architecture pattern. It also shows that the software engineering discipline needs enterprise perspectives to be able to include all aspects of sustainable industrial software system development.

Classification of the USAP artifacts made use of the business concept perspective for four of the twelve artifacts. The inclusion of a traditional enterprise perspective led to new conclusions regarding the use of general activities for pattern creation. General domain application activities and their tasks make use of the domain's role and work product as placeholder. The general activities and tasks then become domain application specific. The reusable task has reusable responsibilities and by specifying what quality attribute the task support, the responsibilities can be constructed to support that specific quality of the task. This has been shown for usability in the USAP study. The USAP information description-selection process could be described by following Zachman's consistency rules in the Software Engineering Taxonomy.

F.1 Introduction

For a software engineering researcher it can be useful to answer journalistic questions regarding the information collected in field studies and case studies. Journalistic abstractions are typically: “What does the information describe?”; “How is the information used?”; “Where is the information used?”; “Who is using the information?”; “Why is the information used?”. Depending on the usage perspective of the information, the answers will differ. If the information is related to the perspective of the system’s development organization, the answers will be different than if the information is related to the perspective of the system’s architecture.

How information from the development organization’s perspective and from the system’s architecture perspective relate to each other could also be helpful to describe. For example, sustainable development of an industrial software system organization is impacted by organizational patterns, architecture patterns and the knowledge transfer in the organization. Conducting a case study exploring sustainable development in the domain of industrial software systems, will collect information from many perspectives. It would then be helpful for software engineering researchers to use a enterprise architecture taxonomy where the journalistic abstractions and the usage perspectives act as classifier of the information.

In previous work we presented a derivative of the Zachman framework called the Software Engineering Taxonomy which is suggested for the classification of software engineering information [1]. The following sections describe how the Software Engineering Taxonomy is applied to three studies: the Usability Supporting Architecture Patterns study [2][3], the Influencing Factors method study [4], and the Sustainable Industrial Software Systems study [5].

F.2 Software Engineering Taxonomy

In a joint article [6], published 1992, Sowa and Zachman explain that the Zachman framework links the concrete things in the world (entities, processes, locations, people, times and purposes) to the abstract bits in the computer. The Zachman framework is not a replacement of programming tools, techniques, or methodologies but instead, it provides a way of viewing the system from many different perspectives and how they are all related. The framework logic can be used for describing virtually anything considering its history of development.

The logic was initially perceived by observing the design and construction of buildings. Later it was validated by observing the engineering and manufacture of airplanes. Subsequently, it was applied to enterprises during which the initial material on the framework was published [7][8][9]. Sowa and Zachman write:

> *Most programming tools and techniques focus on one aspect or a few related aspects of a system. The details of the aspect they select are shown in utmost clarity, but other details may be obscured or forgotten. By concentrating on one aspect, each technique loses sight of the overall information system and how it relates to the enterprise and its surrounding environment. The purpose of the Information System Architecture framework is to show how everything fits together. It is a taxonomy with 30 boxes or cells organized into six columns and five rows. Instead of replacing other techniques, it shows how they fit in the overall scheme.*

According to Zachman, "Architecture" is the set of descriptive representations relevant for describing a complex object (actually, any object) such that the instance of the object can be created and such that the descriptive representations serve as the baseline for changing an object instance.

The columns of the framework represent different abstractions from, or different ways to describe, information of the complex object. The reason for isolating one variable (abstraction) while suppressing all others is to contain the complexity of the design problem. Abstractions classifying the description focus are:

Inventory Sets - Describes "What" information is used

Process Transformations - Describes "How" the information is used

Network Nodes - Describes "Where" the information is used

Organization Groups - Describes "Who" is using the information

Timing Periods - Describes "When" the information is used

Motivation Reasons - Describes "Why" the information is used

The rows of the framework represent "Perspectives" classifying the description usage. The perspectives are:

Abstraction / Perspective	INVENTORY SETS (WHAT)	PROCESS TRANSFORMATIONS (HOW)	NETWORK NODES (WHERE)	ORGANIZATION GROUPS (WHO)	TIMING PERIODS (WHEN)	MOTIVATION REASONS (WHY)
SCOPE CONTEXTS (Strategists)	e.g. Inventory Types	e.g. Process Types	e.g. Network Types	e.g. Organization Types	e.g. Timing Types	e.g. Motivation Types
BUSINESS CONCEPTS (Executive Leaders)	e.g. Business Entities & Relationships	e.g. Business Transform & Input	e.g. Business Locations & Connections	e.g. Business Role & Work	e.g. Business Cycle & Moment	e.g. Business End & Means
SYSTEM LOGIC (Architects)	e.g. System Entities & Relationships	e.g. System Transform & Input	e.g. System Locations & Connections	e.g. System Role & Work	e.g. System Cycle & Moment	e.g. System End & Means
TECHNOLOGY PHYSICS (Engineers)	e.g. Technology Entities & Relationships	e.g. Technology Transform & Input	e.g. Technology Locations & Connections	e.g. Technology Role & Work	e.g. Technology Cycle & Moment	e.g. Technology End & Means
COMPONENT ASSEMBLIES (Technicians)	e.g. Component Entities & Relationships	e.g. Component Transform & Input	e.g. Component Locations & Connections	e.g. Component Role & Work	e.g. Component Cycle & Moment	e.g. Component End & Means

Figure 1: The Zachman Framework

Scope Contexts - perspective descriptions corresponds to an executive summary for a planner or investor who wants an estimate of the scope of the system, what it would cost, and how it would perform.

Business Concepts - perspective is the perspective of the owner, who will have to live with the constructed object (system) in the daily routines of business. This perspective descriptions correspond to the enterprise (business) model, which constitutes the design of the business and shows the business entities and processes and how they interact.

System Logic - perspective is the designer's perspective. The System Logic perspective descriptions correspond to the system model designed by a systems analyst who must determine the data elements and functions that represent business entities and processes.

Technology Physics - perspective descriptions correspond to the technology model, which must adapt the system model to the details of the programming languages, I/O devices, or other technology. This is the perspective

where the four views of the "4+1" model by Kruchten [10] can be used to describe software architecture.

Component Assemblies - perspective descriptions correspond to the detailed specifications that are given to programmers who code individual modules without being concerned with the overall context or structure of the system.

The relevant descriptive representations would necessarily have to include all the intersections between the Abstractions and the Perspectives (Figure 1). "Architecture" would be the total set of descriptive representations (models) relevant for describing the complex object and required to serve as a baseline for changing the complex object once it is described. Zachman's complex object is the enterprise, but principally he states that the complex object can be any object.

The Zachman framework is a structure, not a methodology for creating the implementation of the object. The Zachman Framework does not imply anything about how architecture is done (top-down, bottom-up, etc). The level of detail is a function of a cell not a function of a column. The level of detail needed to describe the Technology Physics perspective may be naturally high but it does not imply that the level of detail of the Scope Contexts descriptions should be lower or the opposite.

The framework is normalized, that is adding another row or column to the framework would introduce redundancies or discontinuities. Composite models and process composites are needed for implementation. A composite model is one model that is comprised of elements from more than one framework model. For architected implementations, composite models must be created from primitive models and diagonal composites from horizontally and vertically integrated primitives. The structural reason for excluding diagonal relationships is that the cellular relationships are transitive. Changing a model may impact the model above and below in the same column and any model in the same row.

The rules of the framework are [8]:

Rule 1: Do not add rows or columns to the framework

Rule 2: Each column has a simple generic model

Rule 3: Each cell model specializes its column's generic model

Rule 3 Corollary: Level of detail is a function of a cell, not a column

Rule 4: No meta concept can be classified into more than one cell

Rule 5: Do not create diagonal relationships between cells

Rule 6: Do not change the names of the rows or columns

Rule 7: The logic is generic, recursive

The model, i.e. the view, in the Zachman framework can be aligned with the ISO/IEC 42010:2007 viewpoints [11]:

> *An organization desiring to produce an architecture framework for a particular domain can do so by specifying a set of viewpoints and making the selection of those viewpoints normative for any Architectural Description claiming conformance to the domain-specific architectural framework. It is hoped that existing architectural frameworks, such as the ISO Reference Model for Open Distributed Processing (RM-ODP) [12], the Enterprise Architecture Framework of Zachman [7], and the approach of Bass, Clements, and Kazman [13] can be aligned with the standard in this manner.*

Zachman's framework does not describe what language to use for the model descriptions or how to do the actual modeling for each cell. Therefore each view of the Zachman's framework is free to use the viewpoint selected by the responsible of the description. It should therefore be possible to use the viewpoints from the ISO/IEC 42010:2007 to describe a model, i.e. a view, within the framework.

For manufacturing a process composite would be necessary. The process composite describes the working process of creating the model descriptions of the composite model, typically ending with the descriptions of the components in the Component Assemblies perspective, e.g. a service or framework. A third dimension of the framework, called science, has been proposed by O'Rourke et al. [14]. This extension is known as the Zachman DNA (Depth iNtegrating Architecture). In addition to the perspectives and aspects the z-axis is used for classifying the practices and activities used for producing all the cell representations.

In order to be able to use the Zachman framework for software engineering artifacts, two basic assumptions were done:

1. The software engineering classification framework, derived from the Zachman framework, describes the software system's development organization and the customer's scope and business related to the need of system support.

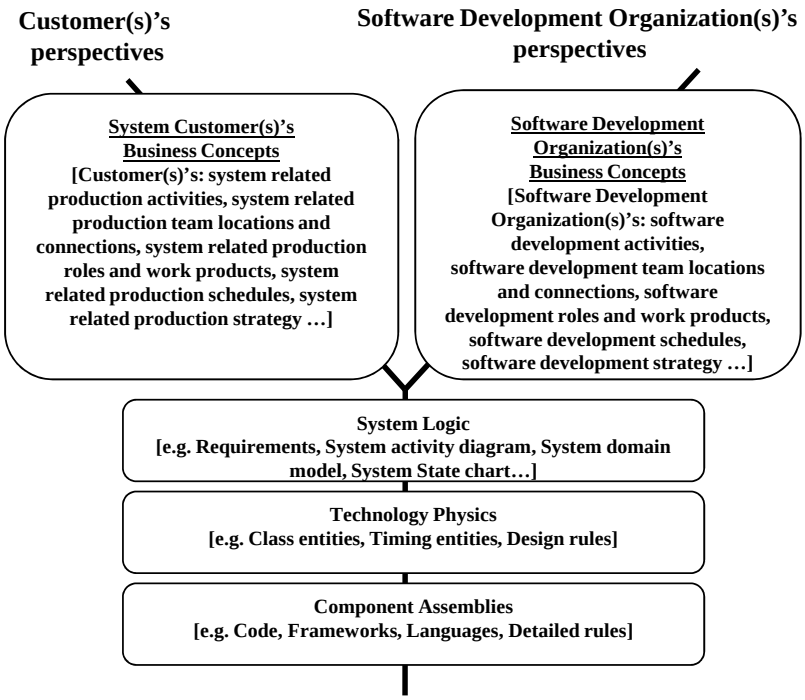

Figure 2: The Customer's and the Software Development Organization's perspectives

2. The software engineering classification framework, derived from the Zachman framework, is three-dimensional where site is the third dimension. The site might be the software development organization, external development organization or the customer's enterprise as long as the site has a part in the system usage or system development.

The assumptions are illustrated in Figure 2. With these assumptions, the system development's Business Concepts perspective will describe the software development artifacts, e.g. software development activities, software development team locations and connections, software development roles and work products, software development schedules, and software development strategies. The models in the customer's Business Concepts perspective will describe the customer's production related to the need of system support. The

resulting software engineering classification framework is called the Software Engineering Taxonomy.

F.3 Software Engineering Taxonomy and System Sustainability

In previous work [5], the presented sustainable industrial software systems theory introduces some insights into the importance of time dynamics for the sustainability of industrial software systems. The time dynamics is discussed not only for technology factors but also for organizational and business related factors, which are enterprise architecture factors. Change of business goals and their co-existence with changes in organization and market environments are also discussed leading to a deeper exploration of a broader spectrum of the enterprise architecture and its relation to system- and software architecture. The case study's units of analysis were companies with the following software development characteristics:

- The company's software development involved at least 20 developers
- The company had software systems with a life-time of 10 years or more
- The company developed industrial automation applications.

From May 2008 through December 2008, three automation system companies with these characteristics were visited. Three roles were interviewed at each company: senior software developer, senior software architect, and senior product manager. The same questions, based on the sustainable industrial software systems theory, were asked to all of the nine interviewees. Structured individual interviews were conducted, which were approximately three hours long, on site. Participants were guaranteed anonymity, and the information reported was sanitized so that no individual person or company could be identified. The questions were open-ended and allowed participants to formulate answers in their own terms. The preliminary case study findings were presented to the participating companies and additional companies in an architecture day workshop where software architects and management were invited to discuss the findings.

F.3.1 Sustainable Industrial Software System Development

Pollan has defined an unsustainable system simply as "a practice or process that can't go on indefinitely because it is destroying the very conditions on which it depends"[15]. Unruh has argued that numerous barriers to sustainability arise because today's technological systems were designed and built for permanence and reliability, not change [16].

"A global agenda for change" - was what Gro Harem Brundtland, as the chairman of the World Commission on Environment and Development, was asked to formulate in 1987 [17]. As a result, the Brundtland commission defined sustainable development as:

> *Sustainable development is development that meets the needs of the present without compromising the ability of future generations to meet their own needs. It contains within it two key concepts: the concept of "needs", in particular the essential needs of the world's poor, to which overriding priority should be given; and the idea of limitations imposed by the state of technology and social organization on the environment's ability to meet present and future needs.*

In [18], Dyllick and Hockerts transpose the definition to the business level:

> *Corporate sustainability is meeting the needs of a firm's direct and indirect stakeholders (such as shareholders, employees, clients, pressure groups, communities etc), without compromising its ability to meet the needs of future stakeholders as well.*

Following the reasoning of the Brundtland commission [17] and Dyllick and Hockerts [18], sustainable industrial software development would be defined as:

> *Sustainable industrial software development meets the needs of the software development organization's direct and indirect stakeholders (such as shareholders, employees, customers, engineers etc), without compromising the organization's ability to meet its future stakeholders' needs as well.*

In this report, the term "corporate sustainability" is used when the work referred to uses the term. Otherwise the term "sustainable development" is used.

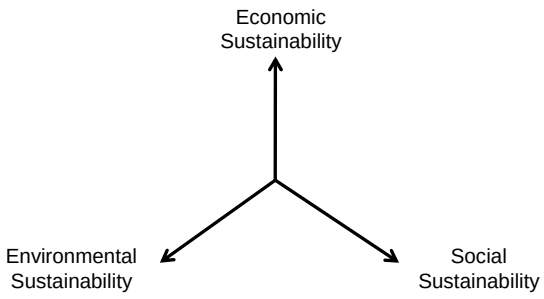

Figure 3: Three dimensions of corporate sustainability

Three dimensions of corporate sustainability is outlined by Dyllick and Hockerts: environmental sustainability, economic sustainability, and social sustainability, the "triple-bottom-line" in Figure 3. Dyllick and Hockerts conclude that a single-minded focus on economic sustainability can succeed in the short-run; however, in the long-run sustainability requires all three dimensions to be satisfied simultaneously.

Sustainable development of industrial software systems is a true challenge due to changes in concerns originating from: new technology, new stakeholder needs, new organizations, and new business goals during decades. It's challenging since it has not been researched for industrial software systems and the domain need an understanding of the success-critical concerns related to the achievement of sustainable development of systems as the complexity of organizations, processes, and architectures increase.

Organizational complexity involves many success-critical stakeholders, often located all over the world, who have to reach a consensus around the most important business goals for the system now and in the next future. Sustainable systems have the built-in legacy heritage and have to consider the present software architecture and design when introducing new business goals. Stakeholders, including the architects, need an understanding of how the organization's business goals affect architectural qualities and vice versa. For example, industrial software systems are often affected by company mergers and acquisitions, where two or more systems have to be consolidated into one system or the systems have to share a core part. The effect of such decision on software quality is hard to overlook. Sustainability is therefore related not only to software structures and their interactions but also to the system's environment

in terms of the enterprise aspects as organization, business, tactics and scope. Enterprise aspects have not been put in relation to software architecture and implementation for industrial software systems in an explicit way earlier. As organizational complexity grows when the systems are distributed developed, the impact of the enterprise aspects on the software system is significant.

F.3.2 Case Study Questions and Propositions

The theory presented in paper [5] was the base for the the planning of a case study intended to investigate the definition of a sustainable industrial software system and the sustainability success-factors of three companies developing sustainable industrial software systems. The case study design followed the proposed design by Yin [19]. The quality of the case study was tested by the four tests suggested by Yin:

Construct Validity: The case study's units of analysis were companies that: involved at least 20 developers; had software systems with a life-time of 10 years or more; and developed industrial automation applications. From May 2008 through December 2008, three automation system companies with these characteristics were visited. Three roles were interviewed at each company: senior software developer, senior software architect, and senior product manager. The same questions based on the theory in Paper **B** were asked to all of the nine interviewees.

Internal Validity: Not applicable since the case study is not a explanatory or causal case study.

External Validity: The domain to which the case study findings can be generalized is the domain of long-lived industrial software systems. The case study's three units of analysis were companies that: involved at least 20 developers; had software systems with a life-time of 10 years or more; and developed industrial automation applications. Comparison of the findings has been made with the theory proposed by Curtis et al. [20][21]. Curtis et al. conducted an extensive field study involving 19 projects in the domain of large complex software systems ranging from aerospace contractors to computer manufacturers with real-time, distributed, or embedded applications. To further strengthen the external validity the case study interview should be conducted with e.g. automotive companies also developing large complex long-lived software systems.

Reliability: Structured individual interviews were conducted which were approximately three hours long on the interviewee's site. Participants were guaranteed anonymity and the information reported has been sanitized so that no individual person or company can be identified. The questions were open-ended and allowed participants to formulate answers in their own terms. One person had the lead as questioner in each interview and one person had the responsibility for taking notes. After the interview the person who had the lead responsibility for taking notes wrote the interview protocol and sent it to the other person for review. Then the lead responsible for taking notes revised the protocol and as a last validation sent the protocol to the interviewee for review. The preliminary case study findings were presented to the participating companies and additional companies in an architecture day workshop where software architects and management were invited to discuss the findings.

The case study propositions were:

1. We believe sustainable systems can control the development cost
2. We believe the customers expect the system to be long-lived
3. We believe that offering a sustainable system is a market advantage
4. We believe that sustainable systems must cope with change in organizations, technology, business goals, and stakeholders' concerns, without losing control over its cost, quality and schedule output
5. We believe sustainable systems will have an organization with a high communication interaction
6. We believe that organizations that manage sustainable systems will have an organization with clearly defined roles and clear hand-over of information
7. We believe that organizations that manage sustainable systems will plan for changes by forward feeding them upon detection into the planning of next major steps of the system
8. We believe that organization that manage sustainable systems will have stated long-term business goals communicated to the entire organization.
9. We believe that major organizational changes are the most difficult changes for a sustainable system

10. We believe sustainable systems can do major architectural changes without the customers noticing any major changes to the product. For instance, migrating to a product-line architecture without changing the essences of the product

11. We believe sustainable systems have high-frequent control over development progress in between release dates

The case study questions were formulated in a way that the answers could provide data to verify or reject the propositions. The case study's question "What is system sustainability to you?" was asked to all of the interviewees to let them define the concept of a sustainable industrial software system. By doing so, the interviewees could relate to their own definition when answering the rest of the questions regarding system sustainability.

F.3.3 Classification of Case Study Data

Concerns related to sustainability were extracted from the answers. When doing so, sustainability concerns were extracted which the interviewees thought they had met in a good way. Additionally sustainability concerns were extracted which the interviewees wanted to meet in a better way because they believed meeting these concerns would improve the sustainability of the system. The resulting concerns were mapped in the Software Engineering Taxonomy with the Scope and Business perspectives being the perspectives of the system development organization. The result of the mapping of the collected data in the Software Engineering taxonomy is shown in Figure 4.

F.3.4 Analysis of Classified Case Study Data

The product managers had exhaustive answers around concerns with Scope Contexts- and Business Concepts perspectives. Surprisingly, the senior developers and the architects did not have the corresponding exhaustive answers around concerns with System Logic- and Technology Physics perspective. This could very well relate to the reported unclear developer role- and architect role descriptions. Further the answers described how the developers and architects did not have documented software architecture, defined software architecture or an architecture design process. The developers and architects, according to the interview answers, simply lack many of the model descriptions from the System Logic perspective and the Technology Physics perspective.

Abstraction / Development Perspective	Inventory Sets (WHAT)	Process Transformations (HOW)	Network Nodes (WHERE)	Organization Groups (WHO)	Timing Periods (WHEN)	Motivation Reasons (WHY)
Scope Contexts	Well-known sustainable key competences --- Well-known key stakeholders; Well documented system knowledge; Sustainable HMI technology; Documented role descriptions	Flexible Project Management Process; Flexible in-house software development process; Formal technology evaluation process; Formal architecture evaluation process	Comply with standardization organizations and federal agencies	Minimal target market competition; --- Sustainable 3d-party software; Sustainable HMI technology vendors; Sustainable development organization groups	Keep track of competitors' releases	Sustainable revenue strategy; Sustainable target markets; --- Open and communicative organization culture
Business Concepts	Short-term based decisions balanced with long-term considerations; Feature-driven and quality-driven ROI; Maintenance cost separated from development cost; Globally applicable development KPIs; Objective time-prediction algorithm for development projects	Excellent technology scouting; Few customer-tailored projects; Quality improvement projects balanced with development projects; Keep close contact with target market customers; Analyze target market needs for new technology	High-frequent communication between 3d party product supplier and development organization --- High-frequent communication between Product Management and architects; High-frequent communication between distributed development teams;		Long system life cycle --- Release cycle, in balance with customer-desired system update-rate; High-frequent project follow-up cycles	Strategy for keeping sustainable key-competences; --- Cultural boundaries communication strategy; Well-communicated system-related customer goals and development goals
System Logic		Don't mimic organizational groups' interfaces when designing system components' interfaces; Minimum of complexity in architecture				Reliability; --- Usability; --- Maintainability; Portability; Modifiability; Scalability, Understandable requirements
Technology Physics	Isolated Business Logic --- Sustainable HMI technology components	Sustainable Business Logic supporting sustainable customer business processes	Stable system interoperation interfaces,	Low-frequent changing HMI		
Component Assemblies	Re-usable components		Standardized communication protocols			

Figure 4: The Enterprise-wide concerns related to corporate sustainability: The check signs indicate that the concerns are met by the companies; The warning signs indicate that the companies want to meet the concerns in a better way

Even if the term software engineering was coined as early as in the 1968 NATO Software Engineering Conference [22] and Dijkstra described software structures the same year [23], the usage of software engineering and software architecture concepts and tools in the domain of industrial software systems is low.

Basili and Musa write that *"...we must isolate and categorize the components of the software engineering discipline, define notations for representing them and specify the interrelationships among them as they are manipulated"* [24]. Jackson claims that: *"...there will never be software engineering. As these specializations flourish (e.g. compiler engineering, operating systems [author's remark]) they leave software engineering behind ...A professor of software engineering must, by definition, be a professor of unsolved problems"* [25]. There is an unclear definition of what software engineering is and what the important components of the software engineering discipline are. Industrial software system organizations lack clear guidance on what kind of descriptions would give the best return of investment in their domain. One questions asked in the case study was "What is a major architectural change?" to let the interviewees describe their perception of architecture and changes to it. The answers varied from the question being an philosophical question to an architectural rule change. But no two persons' answers were the same.

According to Garlan and Shaw [26], the definition of software architecture is:

> *software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns*

In [13], Bass et al. define software architecture as:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

According to Gacek et al. [27], a software system architecture comprises:

- A collection of software and system components, connections, and constraints.
- A collection of system stakeholders' need statements.

- A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.

Johnson has investigated the definitions of software architecture to find a general consensus among the definitions [28]. But Johnson resorts to conclude that *"It is not generally agreed upon what a component or entity is, it is not generally agreed upon what a structure is, or even if it is to be called structure, and it is not generally agreed upon what else comprises software architecture"*.

Considering Johnson's conclusion, the question is how the differences in agreement upon what comprises software architecture affect a not risk-willing industry's adaptation of software architecture's concepts. When each industry or application area has to define its own understanding of the meaning of software architecture, it might lead to that traditional software-intensive domains take a lead in the adaptation of software architecture concepts and the non-traditional software-intensive domains have a long way to go to reach the same software quality maturity. If software quality maturity affects the sustainability of the software system, this is a serious issue without an obvious solution. Each software application domain can hardly define its own software engineering research discipline as Jackson discusses [25].

The case study questions were analyzed to find out if something had been missed that would have scattered some light on the absent software architecture concerns. However, the interview contained several questions related to the relation between architecture and technology for system sustainability. It seems like the case study's findings confirm Curtis's reasoning. Curtis writes that the software production efficiency is not a function of only software engineering methods and quality thinking but to a larger extent a function of organizational issues such as behavior and communication [21].

Additionally one could speculate in if the lack of model descriptions from these perspectives in itself is a sustainability concern. According to the interview answers this is the case. The lack of system documentation is mentioned by all roles at all companies as a hinder for corporate sustainability. One conclusion could be that in order to get the software engineering process artifacts, e.g. architecture descriptions, in place the companies must get the organizational artifacts, e.g. role descriptions and communication, in place first. Curtis's study and the System Sustainability case study point toward a possible conclusion that a working software development organization, with model descriptions from the Business Concepts perspective in place, is a prerequisite for software engineering tools and methods to have a significant impact on

productivity and sustainability.

In [29], Malveau and Mowray suggest a Software Design-Level Model (SDLM):

> *The Software Design-Level Model (SDLM) builds upon the fractal model. This model has two major categories of scales: Micro-Design and Macro-Design. The Micro-Design levels include the more finely grained design issues from application (subsystem) level down to the design of objects and classes. The Macro-Design levels include system-level architecture, enterprise architecture, and global systems (denoting multiple enterprises and the Internet). The Micro-Design levels are those most familiar to developers. At Micro-Design levels, the key concerns are the provision of functionality and the optimization of performance. At the Macro-Design levels, the chief concerns lean more toward management of complexity and change. These design forces are present at finer grains, but are not nearly of the same importance as they are at the Macro-Design levels.*

Using the concepts of the Software Design-Level Model, the collected interview data suggest that the interviewees have a vast majority of sustainability concerns at the Macro-Design level, described in the Software Engineering Taxonomy's Scope Contexts perspective and in the Business Concepts perspective. Management of complexity and change are tightly coupled to sustainability concerns [5].

In the Pasteur research project at Bell Labs [30], Coplien et al. investigated organizational structures. Coplien's organizational studies found two organizational patterns:

- Architecture Follows Organization, a restatement of Conway's Law [31].
- Organization Follows Location, no matter what the organizational chart says.

A discussion related to Coplien's first organizational pattern with one architect in the Sustainable Industrial Software System case study was about what was the best alternative; to let the organization decide the architecture or to let the architecture decide the organization.

Cain et al. have described additional organization patterns [32]. Their conclusion is that: "If there is one consistent measure of successful organization, it is how well its members maintain relationships through communication".

Dikel et al. developed organizational principles in an effort to predict the success or failure of software architectures for large telecommunications systems [33]. In the reported case study [33], they realized that technical factors, do not by themselves explain the success of a product-line architecture and that only in conjunction with appropriate organizational behaviors can software architecture effectively control project complexity. The view of the software architecture as a control instance working correctly only if the organizational parameters are set correctly led Dikel et al. to reflect on the law developed by Ashby [34], the *law of requisite variety*, which suggests that a system should be as complex as its environment:

> *. . . in active regulation only variety can destroy variety. It leads to the somewhat counterintuitive observation that the regulator must have a sufficiently large variety of actions in order to ensure a sufficiently small variety of outcomes in the essential variables E. This principle has important implications for practical situations: since the variety of perturbations a system can potentially be confronted with is unlimited, we should always try maximize its internal variety (or diversity), so as to be optimally prepared for any foreseeable or unforeseeable contingency.*

Dikel et al. reason around that if a software architecture becomes more complex than its environment, it may become too expensive for the organization to support. In the book [35], Kane et al. describe 30 organizational patterns and anti-patterns using the principles; Vision, Rhythm, Anticipation, Partnering and Simplification (VRAPS).

If the environment would include the organizational environment as well as the business environment then both the Micro-Design level [29] patterns (discussed by Beck [36], Buschman [37] , Shaw [26], Gamma [38] and Fowler [39]) as well as the Macro-Design level [29] patterns (discussed by Fowler [39], Coplien [40] and Kane [35]) must harmonize in their complexity with the complexity of the software architecture for a sustainable software system. For industrial software systems, a domain model of the business domain along with a measure of its complexity would be required in order to understand on what level the software architecture complexity should be.

Many attempts of measuring software architecture complexity have been made: Boehm et al. describe MBASE that considers architectural complexity [41]; Halstead [42] proposes measures to predict understanding effort based on grammatical complexity of code modules; McCabe [43] proposes a graph-theoretic cyclomatic complexity measure etc. The question is if, and in that

case, what kind of organizational and architectural complexity measure should be used in the law of requisite variety if it were to be applied to software engineering for the sustainability of industrial software systems.

In the following lists of the sustainability concerns, the concerns' importance for sustainability is ranked. The ranking is done according to how many of the interviewees mentioned the concern as important for sustainability or desirable for sustainability. If four or more interviewees mentioned the concern, then it got ranked as ***; if two or three interviewees mentioned the concern, then it got ranked as **; and if only one interviewee mentioned the concern, then it got ranked as *.

Concerns with Scope Contexts perspective:

1. Inventory Sets Abstraction
   (a) Well-known sustainable key competences***
   (b) Well-known key stakeholders*
   (c) Well documented system knowledge***
   (d) Sustainable Human Machine Interface (HMI) technology*
   (e) Documented role descriptions***
2. Process Transformations abstraction
   (a) Formal in-house software development process*
   (b) Formal technology evaluation process***
   (c) Formal architecture evaluation Process***
3. Network Nodes Abstraction
   (a) Comply with standardization organizations and federal agencies***
4. Organization Groups abstraction
   (a) Sustainable standards***
   (b) Sustainable 3d party software***
   (c) Sustainable HMI technology vendors*
   (d) Sustainable development organization groups***
5. Motivation Reasons abstraction

(a) Sustainable revenue strategy*

(b) Sustainable target markets in need of sustainable systems***

(c) Open and communicative organization culture***

It's striking that so many concerns with a Scope Contexts perspective are seen as having high importance for corporate sustainability. Not all of these concerns are targets for traditional software engineering but many of them actually are, such as: stakeholders, documented system knowledge, software development process, and architecture evaluation process. Other concerns are dealt with within the field of organizational theory: key competences, role descriptions, project management process, development organization groups, and organization culture. Some are related to the field of economics: revenue strategy, target markets. Some concerns are related to technology: HMI technology, technology evaluation process, 3d party software, HMI technology vendors and standardization organizations. Compliance with federal agencies' regulations processes may be a cross-cutting concern.

Concerns with Business Concepts perspective:

1. Inventory Sets abstraction

   (a) Short-term and long-term gain in balance in cost-benefit analysis***

   (b) Feature-driven and quality-driven Return Of Investment calculation***

   (c) Maintenance-phase cost separated from design-phase cost**

   (d) Globally applicable development Key Performance Indicators (KPIs)**

   (e) Objective time-prediction of software development tasks***

2. Process Transformations abstraction

   (a) Excellent technology scouting***

   (b) Few customer-tailored architectural changes***

   (c) Quality improvement projects balanced with feature development projects***

   (d) High-frequent communication between target market customers and product managers***

   (e) Analysis of target market need of new technology***

3. Network Nodes abstraction
   (a) High-frequent communication between 3d party product supplier and development organization***
   (b) High-frequent communication between product management and architects***
   (c) High-frequent communication between distributed development teams***
4. Organization Groups abstraction
5. Timing Periods
   (a) Long system life cycle***
   (b) Release cycle in balance with customer-desired system update-rate***
   (c) High-frequent project follow-up cycles*
6. Motivation Reasons abstraction
   (a) Strategy for keeping sustainable key-competences***
   (b) Cultural boundaries communication strategy***
   (c) Well-communicated system-related customer goals and development goals*

Sustainability concerns with a Business Concepts perspective are seen as having high importance by all roles. The Business Concepts perspective in the Software Engineering Taxonomy, mapping the case study data, is the business perspective of the development organization and deals with everyday work issues for all people working in the development organization. The interviewees, with the exception of one architect, had all worked for 10 years or more within their current organization and in this time they had collected Business Concepts concerns they see as highly important for the sustainability of the system they develop.

The Inventory Set perspective's mapped concerns have influences from software engineering-, economics-, and management theory. The interviewed product managers asked for better ways of calculating the Return Of Investment for quality-focused projects and for long-term projects. The current calculations benefit feature-driven projects as well as short-term projects resulting in developers hiding quality-improvement they see as necessary in the feature-driven projects. This could be one reason for over-optimistic time-prediction calculations done by the developers, since they only get approval for feature

implementations. However, calculating a correct development effort for a proposed change request is difficult.

Curtis describe the time required for learning application-specific information as being buried under the traditional life cycle phase structure of most projects and unaccounted for [20]. Thus, Curtis continues, the time required to create a design is often seriously underestimated. By including the educational aspect into the development effort estimations, the estimation might be more correct than today. Some of the interviewees reported on expert developers making better estimations than non-expert developers. The expert developer had long-time experience of the system and probably of the application domain of the customers as well. These expert developers hence would need less education effort than the others, contributing to making their time-estimates more correct.

None of the interviewees had a clear picture of how they measured schedule alignment and development efficiency. The Key Performance Indicators (KPIs) mentioned was the number of System Problem Reports related to quality-in-use. The SPRs are reported by customers and testers. The product managers said they would like to see a globally applicable KPIs that measures development performance in distributed development teams. Separating maintenance cost from design cost would be a prerequisite for the use of a globally applicable KPI since maintenance and design have different characteristics. According to the IEEE 610.12-90 definition [44], adopted by the IEEE Software Engineering Book Of Knowledge (SWEBOK) [45], design is both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process". SWEBOK describes software maintenance as "Once in operation, anomalies are uncovered, operating environments change, and new user requirements surface. The maintenance phase of the life cycle commences upon delivery".

Globally applicable KPI could be based on the categories of identified information needs in the development organization suggested by Antolic [46]: Schedule and Progress; Resources and Cost; Product Size and Stability; Product Quality; Process Performance; Technology Effectiveness; Customer Satisfaction. The KPIs could also be based on the complexity measures discussed in: Boehm et al. [41], Halstead [42] or McCabe [43].

The customer-specific architectural change projects were a sustainability concern voiced by all developers and architects. This confirms the top-two finding, in Curtis's study, related to fluctuating requirements as a hinder for software development productivity [21]. One architect in the Curtis's study said:

> *Software architect: The whole software architecture, to begin with, was designed around one customer that was going to buy a couple of thousand of these. And it was not really designed around the . . . , marketplace at all . . . Another . . . , customer had another need, so we're, trying to rearrange the software to take care of these two customers. And when the third one comes along, we do the same thing. And when the fourth one comes along, we do the same thing.*

A similar statement was voiced by some of the interviewed developers and architects. This does not necessarily have to be a bad thing if the software system is designed to have configuration possibilities for tailoring the system for a specific customer. But for the system to be designed this way, the target marketplace most important business processes have to be known and the system designed around these. Coplien has suggested the domain analysis as one way of finding commonalities for a system's target market [47]. This relates to the sustainability concern findings: "Keep close contact with target market customers" and "Analyze target market needs for new technology".

Concerns with System Logic perspective:

1. Process Transformations abstraction
   (a) Don't mimic organizational groups' interfaces when designing system components' interfaces*
   (b) Minimum of complexity in architecture**
2. Motivation Reasons abstraction
   (a) Reliability***
   (b) Usability***
   (c) Maintainability***
   (d) Portability**
   (e) Modifiability**
   (f) Scalability**
   (g) Understandable requirements**

Maintainability of the system is crucial for customers and developers. Since the system is an expensive long-term investment for both developer and customer, the maintenance phase is very long ranging from ten to thirty years.

Portability, modifiability, scalability and maintainability are seen as important qualities to achieve. At the same time these qualities are concerns that the companies in the study have difficulties to implement in their systems. Portability, modifiability, scalability and development maintainability are not observable in runtime and are quality concerns that the development organization have. The customers' concerns are related to run-time observable qualities as reliability, usability and maintainability in form of e.g. on-the-fly upgrades and easy integration of inter operating systems. The reliability quality is seen as achieved by the case study's participating companies' interviewees. The development organization's quality concerns not observable in runtime are seen as not fully achieved.

Concerns with Technology Physics perspective:

1. Inventory Sets abstraction
   (a) Isolated Business Logic***
   (b) Sustainable Human Machine Interface (HMI) technology components*
2. Process Transformations abstraction
   (a) Sustainable Business Logic supporting sustainable customer business processes***
3. Network Nodes abstraction
   (a) Stable system inter-operation interfaces***
4. Organization Groups
   (a) Low-frequent changing HMI*

In the interviews, the importance of isolating the core business logic from frequent change impact was mentioned several times. The core business logic is a market differentiator and sustainable since it supports the customer process needs that are sustainable. Since these sustainable needs of the customers do not change over decades, the business logic handling these needs is especially important to identify, master and isolate.

The "Stable system inter-operation interfaces" concern was identified as growing in importance due to the growing requirement on interoperability internally at the customer location through intranets and the Internet.

All of the interviews testified that the Human Machine Interface was the part of the system with the most frequent changes. Only one interviewee expressed a desire for sustainable HMI components which could support easy updates to the HMI. This was a bit surprising. If the HMI is the subsystem with the most frequent changes then the concern would logically be to find HMI technology that is sustainable in order for the frequent changes to be less challenging. Relating to the Usability-Supporting Architecture Patterns study of the interplay between usability and software architecture, isolating the user interface logic is not enough to achieve a usable system [2][3]. Architectural changes are necessary in order to support aspects of usability. Frequent changes to the user interface would hence correlate to some changes in the architecture in order to get the desired behavior of the user's interaction with the system. Architectural changes are expensive since an architectural change in a complex legacy system has a series of consequences for the system. The awareness of the interplay between usability and software architecture is however low in the software engineering community. In the IEEE Software Engineering Body Of Knowledge (SWEBOK) published 2004 [45], the word usability is mentioned six times but SWEBOK refers to the software ergonomics discipline for how to work with usability. Rozanski and Woods suggest the isolation of user interface logic as the only usability tactic, in contrast to their thorough descriptions of ten security tactics [48].

Concerns with Components Assemblies perspective:

1. Inventory Sets abstraction
    (a) Re-usable components*
2. Network Nodes abstraction
    (a) Standardized communication protocols***

The issue with re-usable components was a concern for only one of the interviewees. In [49], Jacobson et al. discuss the reuse of components and say that reuse is hard because the following factors have to be interwoven and mastered:

- Vision
- Architecture
- Organization and the management of it

- Financing
- Software engineering process

According to the analysis of data in this case study, there seems to be a lack of long-term quality investments possibly due to the KPI numbers and NPV calculations favoring short-term investments. Only one interviewee saw re-usable components as important for sustainability and this could be due to the difficulty of integrating the re-usability factors, listed by Jacobson, in the software development organization. Another reason might be the lack of software engineering insights among the system's management as discussed in Section F.4. If the management do not involve themselves into the software architecture tactics for how to address maintainability and modifiability concerns, which typically result in long-term investments, the projects with this type of agenda suggested by architects and developers have less chance of being approved and prioritized.

In [18], Dyllick and Hockerts describe the non-balance of short-term needs and long-term needs when setting business goals as:

> *In recent years, driven by the stock market, firms have tended to overemphasize short-term gains by concentrating more on quarterly results than the foundation for long-term success. Such an obsession with short-term profits is contrary to the spirit of sustainability, which requires a balance between long-term and short-term needs, so as to ensure the ability of the firm to meet the needs of its stakeholders in the future as well as today.*

Case Study Propositions versus analyzed Data

The status of the propositions in relation to the analyzed collected data is:

1. We believe sustainable systems can control the development cost
   (a) This proposition was not verified nor rejected. The interviewed persons were not the ones who controlled the development cost. The case study should have included line managers and project leaders to test this proposition.
2. We believe the customers expect the system to be long-lived

   (a) This proposition is verified. Sustainable system customers do not want unnecessary updates to the system for long time periods, typically 2-3 years. A replacement of the system is accepted with a time-period of typically 10-30 years.

3. We believe offering a sustainable system is a market advantage

   (a) This proposition is verified. Developing a sustainable industrial software system is extremely expensive. Due to the cost, it's very difficult to get a fast Return-Of-Investment when introducing a new system. Not many competitors are willing to take the risk. Additionally the established sustainable system has a market differentiator of being reliable for long times. By being reliable for long times, the system appeals to potential customers not willing to take the risk of investing in a relatively new system on the market.

4. We believe sustainable systems must cope with change in organizations, technology, business goals, and stakeholders' concerns, without losing control over its cost, quality and schedule output

   (a) This proposition is not verified nor rejected. The interviewed persons were not the ones who controlled the development cost, quality and schedule. The case study should have included line managers and project leaders to test this proposition.

5. We believe sustainable systems will have an organization with a high communication interaction

   (a) This proposition is verified. The implicit knowledge of the well-known sustainable key-competences is communicated frequently through informal information channels, e.g. ad-hoc face to face discussions.

6. We believe organizations managing sustainable systems will have an organization with clear defined roles and clear hand-over of information

   (a) This proposition is rejected. The roles of the interviewed persons were not clearly defined and no clear hand-over of information took place. The reason why the development still worked was to find in the implicit knowledge owned by a set of sustainable key-competences in each company. The long work experience gave them an implicit role as a source of information to whom others turned for help when needed.

7. We believe organizations managing sustainable systems will plan for changes by forward feeding them upon detection into the planning of next major steps of the system

   (a) This proposition is verified. When detecting major technology changes, e.g. Visual Basic support with-drawn from Microsoft, the organizations plan for the exchange. The planned steps were pre-studies, architectural planning and release planning. However, when out-sourcing development work to low-cost countries, the organization did not do any pre-studies, or set up any remote conferencing facilities, or gave any courses in distributed work management. The non existent planning of the new distributed work organization was reported as the most major threat to the sustainability of the system by all three companies in the case study.

8. We believe organizations managing sustainable systems will have stated long-term business goals communicated to the entire organization.

   (a) This proposition is rejected. No one of the interviewees could list the most important long-term business goals. They also did not feel that this was a hinder for the system's sustainability.

9. We believe major organizational changes are the most difficult changes for a sustainable system

   (a) This proposition is verified. All interviewees reported on the distributed development organization as the largest threat to system sustainability. Additionally, it was reported on the unclear decision authority the development organization experienced when controlled by more than two organizations located in different parts of the world. The unclear decision authority often led to some kind of consensus decision not optimizing the system but taken to be politically correct.

10. We believe sustainable systems can do major architectural changes without the customers noticing any major changes to the product. For instance, migrating to a product-line architecture without changing the essences of the product

    (a) This proposition is verified. All of the interviewees reported on the importance of backward compatibility and the customers wanting

no unnecessary production stops due to system maintenance. The development organizations planned for architectural changes with the requirement on backward compatibility in focus. At the same time this requirement was perceived as one of the most difficult to achieve causing high development costs. But all interviewees reported that the backward compatibility was a key-market differentiator and as such very important.

11. We believe sustainable systems have high-frequent control over development progress in between release dates

    (a) This proposition was neither verified nor rejected. The interviewed persons were not the ones who controlled the development progress. The case study should have included line managers and project leaders to test this proposition.

Sustainable Development Dimensions

The list of success-critical concerns from the interviews are translated into sustainability capital according to the three dimensions; Economical, Environmental, and Social. Two of the systems support customers' business processes' efforts to reduce energy consumption. Considering the environmental sustainability, the systems therefore help the customer to reduce the consumption of natural energy resources. This support is listed as environmental capital. Additionally, all three companies have good reputation among customers for having a reliable, high-quality product. The reputation is therefore added as an intangible economical capital. Long market presence is one key aspect to the sustainability of the industrial software systems. By having long market presence and a reliable system, the customers trust the system and therefore feel that they take a smaller risk by investing in the system. The target market of the industrial software system is sustainable itself, which make the target market customers willing to invest in a comparably expensive system. These customers feel that they will achieve a return-of-investment in a relative short time compared to the lifetime of their business processes. The sustainable target market is added as tangible economical capital. Due to the high initial development cost of the industrial software system, few competitors are entering the target market since the systems are sold mainly due to long market presence and good reputation. Newcomers have no long target market presence and have not yet built up the good reputation of being reliable for decades. The few competitors on the target markets is also added as economical capital.

Figure 5 shows the distribution of sustainable development capital for the three industrial software system development organizations in the case study. Even if many capital units are classified as economical capital and only one unit as environmental capital, the number of capital units does not say anything about their relative value to the stakeholders. It might be that the single environmental capital unit is more worth to the system's stakeholders than ten of the economical capital units.

There is no balance in the dimensions, the tangible economical sustainability is over-represented. It shows that, for individuals, working in the industrial software system domain, it will take substantial time before the concept of sustainable development will be natural in all of its dimensions. Creating economical value is important for industrial systems, but the sales for two of the systems would not be as high if the systems did not contribute to a reduction of the natural resource consumption. The environmental sustainability is interacting with the economical sustainability. The social sustainability capital was decreased when distributed development was introduced in the companies. Distributed development is seen as the most major threat to the sustainable development. There are ways to make distributed development work and many of them represent an increase in social capital by socialization. Oshri et al. argue that, in order to achieve successful collaboration, firms should consider investing in the development of socialization despite the constraints imposed by global distribution [50]. The socialization efforts could be e.g. increased communication through virtual Face to Face (F2F) meetings, kick-of meeting, progress meetings etc.

F.3.5 Summary

Using the Software Engineering Taxonomy to classify the concerns collected from the interviews, clarified the enterprise architecture perspectives of the concerns, i.e. if the concern was a system architecture concern or an business concepts concern. Most of the concerns were classified in the perspectives where executive leaders and strategist are responsible for the model descriptions. Management of business processes, strategies, risk analysis, external partnerships, communication, staff, target markets etc is seen as the key to achieve sustainable development.

The results of the sustainable industrial software systems case study are: a set of success-critical concerns for sustainability; 5 verified propositions, 2 rejected, and 4 still to be verified or rejected. The list of success-critical concerns does not include as many architectural success-factors as expected. In

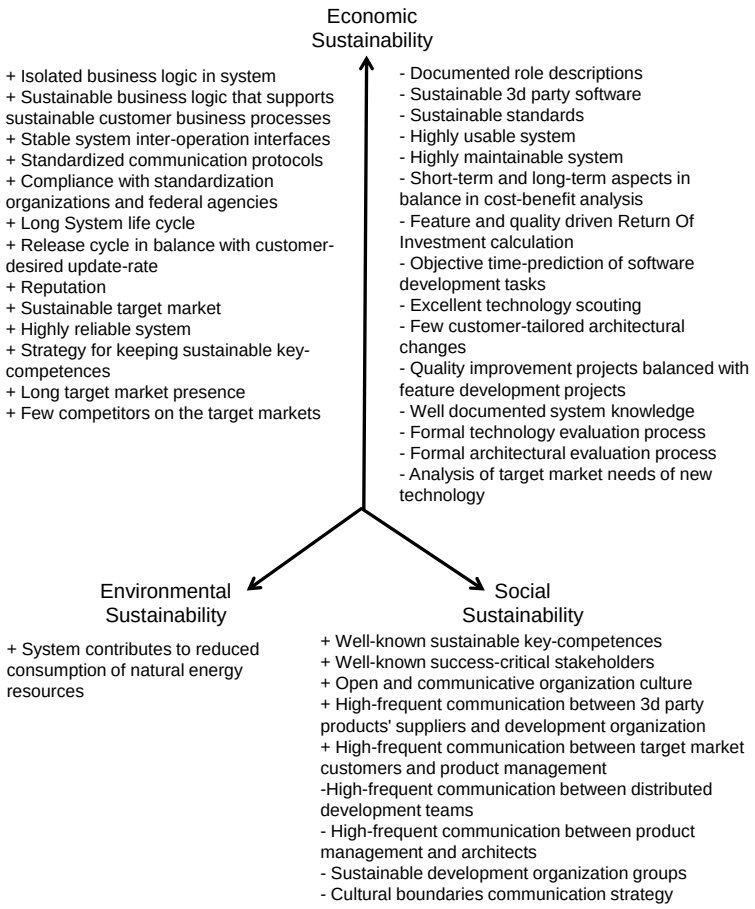

Figure 5: Three dimensions of important sustainable development capital in the domain of Industrial Software System according to the findings. The “plus” sign indicates that the companies felt they had the capital. The “minus” sign indicates they felt they needed an improvement.

the report, it's speculated if this is related to the lack of consensus around the concept of software architecture. The lack of a clear software architecture definition and the lack of tools and methods based on such definition might make the industry reluctant to embrace the concept of software architecture. As long as the software architecture concepts are not explicitly defined, employing software architecture concepts might constitute a risk to the industrial software system development organization. Curtis's study [20][21], Kane's study [35], and the System Sustainability case study point toward a possible conclusion that a working software development organization, with model descriptions from the Business Concepts perspective in place, is a prerequisite for software engineering tools and methods to have a significant impact on productivity and sustainable development.

When applying the concept of sustainable development to the classified concerns from the interviews, which were ranked as being of high importance to the interviewees, there was an unbalance between the economical sustainability, environmental sustainability, and the social sustainability. Most of the concerns addressed economical sustainability or ways of increasing economical sustainability. Some addressed social sustainability but non addressed environmental sustainability. In the analysis, one environmental sustainability issue is added based on knowledge of the systems collected through documentation and experience. When the value of addressing the individual sustainability concern is not known, it's difficult to verify, based on the interrelationships between sustainability dimensions, if the system development is sustainable or not.

F.4 Software Engineering Taxonomy and the IF method

The Influencing Factors (IF) method collects concerns, extracts Influencing Factors from the concerns, and analyzes those for their influence on business goals and software quality attributes. The result is a business goal oriented prioritization of software quality attributes. In [4], the Influencing Factor is a factor that states a motivation for possible system requirements from the stakeholders' perspective.

By presenting the collected effect of several concerns, e.g. in a matrix format [4], the Influencing Factors method makes both the business goal prioritization and the software quality attribute prioritization clear and therefore guides the architectural decisions and strengthens the stakeholders consensus around prioritized concerns. The analyzed concerns could also contribute to a more complete requirement specification, helping the system developers understand the origins of the requirements.

Different impacts of the Influencing Factors are used to prioritize among the Influencing Factors for two authentic cases [4]. The first case was performed on the upgrade of a large legacy industrial software system and the second case on the re-factoring of an existing industrial software system. The two field study systems had a diverse set of stakeholders, such as software architect, system architect, developers, testers, product management, line management, engineers, and users. Both systems suffered from an unclear understanding of what concerns were the most important. The resulting impact analysis helped the stakeholders prioritize among software quality attribute scenarios in the case with the re-factored system. The prioritization included usability and led to the Usability-Supporting Architecture Pattern study [2][3]. The other case, with the legacy system, resulted in the stakeholders' understanding of their perhaps too high focus on short-term market expansion instead of a balanced focus including long-term quality enhancements. Today this company is doing a major investment in enhancing the maintainability of the system.

Influencing Factors from the Influencing Factors case study are here used for additional investigation using the Software Engineering Taxonomy as a reasoning framework. The Influencing Factors are classified in the Software Engineering Taxonomy to explore the possibility of a relation between the classified Influencing Factors and their perspective and abstraction in the taxonomy.

F.4.1 Classification of Influencing Factors

The Influencing Factors are all classified as having the Motivation Reasons abstraction since they describe stakeholder motivations for the usage perspectives: Scope Contexts, Business Concepts and System Logic. Figure 6 shows the classified influencing factors with business goals ownerships and quality attribute impact. The business goal ownership states if it's the customer or development organization that owns the business goal, i.e. has a benefit of achieving the goal. Indirect, the development organization has a benefit of fulfilling the customer's business goals. But the customer business goal would not be addressed by the development organization if the customer had not voiced the goal or concern related to the goal.

F.4.2 Analysis of Classified Influencing Factors

Influencing factors with a System Logic perspective do not have development organization's quality concerns, e.g. testability and maintainability. These concerns never surfaced as part of the success-critical stakeholders' concerns. Testability and maintainability are non-runtime observable qualities [13]. If successfully implemented, the qualities could contribute to long-term cost-reductions for the development organization. However, these two qualities are left to the architect to deal with and take informal decisions on in the investigated cases.

The understanding and interest to deal with software tactics to implement non-runtime observable qualities as testability and maintainability seem to be non-present among the success-critical stakeholders. This was verified for the second case in the case study. Runtime observable qualities affecting the customers' perception of the system engage the success-critical stakeholders more.

Non-runtime observable qualities as testability and development maintainability will likely never be voiced by customers and customer responsible persons. It should be noted that the system's operation environment's maintainability concerns, e.g. installation and on-the-fly upgrades, differ from the development environment maintainability concerns.

In the "System Sustainability" study described in Section F.3 the architects and senior developers testified to how difficult it was to build the business case motivating development-environment maintainability improvements projects with a short-term cost-increase and long-term cost savings. One of the findings was that all the companies in the study wanted a cost-benefit calculation method that balanced short-term gains and long-term gains as they felt the

Abstraction → / Software Development Organization Perspective ↓	MOTIVATION REASONS (WHY)		
SCOPE CONTEXTS		Business Goal Ownership	Quality Concern
	IF3.1: Maintain backward compatibility	Customers	Modularity
	IF4.1: Replace in house developed electronics and/or software with standard HW/SW without affecting availability	Developments	Availability
	IF4.7: Decrease development time by introducing the product line system	Developments	Maintainability
BUSINESS CONCEPTS	IF1.2: Implement same performance as today	Customers	Performance
	IF2.1: Make commissioning easier	Customers	Maintainability
	IF2.2: Implement remote access	Customers	Security
	IF2.3: Make it possible to upgrade parts of or whole system easy and fast.	Customers	Maintainability
	...	...	...
	IF3.2: Implement same robustness/availability as today	Customers	Availability
	IF3.3: Implement same accuracy as today	Customers	Performance
SYSTEM LOGIC	IF1.3: Implement fast extensive communication infrastructure.	Customers	Performance
	IF5.2: Handle analogue signals from external system	Customers	Interoperability
	...	...	...

Figure 6: Influencing Factors classified in the Software Engineering taxonomy. The Influencing Factors related Business Goal ownership and Quality Attribute impact are shown next to the classification in order not to clatter the figure. Quality attribute concerns are classified in the System Logic/Motivation Reasons cell.

current calculations much favored the short-term gains.

F.4.3 Summary

The classification of the Influencing Factors into the Software Engineering Taxonomy contributed to some additional observations regarding stakeholder role and stakeholder perspective. For the stakeholders with the Business Concepts perspective, maintainability and testability are handled with software development improvement strategies, e.g. introduction of product lines. The architec-

tural structures for realizing these strategies are seldom discussed among the success-critical stakeholders. Decisions regarding architectural structures are taken informally by the architects. According to the report's case study analysis of sustainable software development, the architects find it hard to build the business case motivating development-environment maintainability improvements projects with a short-term cost-increase and long-term cost savings. The classification of Influencing Factors in the software engineering taxonomy confirmed that this is a problem that has to be addressed e.g. in term of an improved short-term versus long-term gain return of investment calculation.

F.5 Software Engineering Taxonomy and the USAP study

Usability and its interplay with software architecture was discussed in the Influencing Factors paper [4], as one of five quality attributes. In [2][3], the Usability-Supporting Architecture Pattern field study is described and discussed. The field study was done in the domain of industrial software systems.

The field study contributes with a description of an enhanced USAP, three described USAPs according to the enhancements, and a USAP software tool that visualizes the USAP information.

Visualizing the responsibilities in a tool helps the software architects (on a detailed design level) to implement usability support in the software architecture for specific usability scenarios early in the software design phase. The usability design is part of the enterprise architecture, system architecture, and software architecture but has not been put in relation to these in an explicit fashion before. This field study's research has therefore contributed to fill a gap not covered by existing literature in a sufficient way.

The contribution is significant since very few studies can report on software architects being able to use a tool early in the software design in a way that helps them implement usability support in the software architecture. The two architects in the field study used the tool for six hours and reported on a development cost saving of more than five weeks gained by their interaction with the tool.

In this section Software Engineering Taxonomy (SET) will be used in order to create two process composites (methods). The work flow of creating these process composites guided by the SET will be:

1. Identify artifacts of the Usability Supporting Architecture Pattern concept

2. Classify artifacts in the Software Engineering Taxonomy

3. Create a process composite in the Software Engineering taxonomy, by relating the classified artifacts in a sequence adhering to the Zachman laws

The first process composite will describe a sequence for viewing USAP artifacts in order to evaluate a software architecture against the USAPs. The second process composite will describe a sequence of creating USAP artifacts.

F.5.1 USAP Artifact Identification

USAP Responsibility

The word "responsibility" has been used in the publications of USAP [2][3] but not formally defined in the context of the USAP. The responsibility is originally a section of a Class Responsibility Collaborator (CRC) card. CRC cards are used as a brainstorming tool in the design of object-oriented software. The CRC cards were proposed by Cunningham and Beck [51]. They describe responsibilities as:

> *Responsibilities identify problems to be solved. The solutions will exist in many versions and refinements. A responsibility serves as a handle for discussing potential solutions. The responsibilities of an object are expressed by a handful of short verb phrases, each containing an active verb. The more that can be expressed by these phrases, the more powerful and concise the design. Again, searching for just the right words is a valuable use of time while designing.*
>
> *(p. 2 [51])*

The responsibility as described by Beck and Cunningham was later used by Buschmann et al. to describe the responsibilities of classes in architectural patterns [37]. They describe the responsibility as:

> *Responsibility: The functionality of an object or a component in a specific context. A responsibility is typically specified by a set of operations.*
>
> *(p. 438 [37])*

Wirfs-Brock uses responsibilities in the same sense as Beck and Cunningham [52]. She defines the responsibility as:

> *A responsibility = an obligation to perform a task or know information*
>
> *(p. 3 [52])*

Often there is confusion about the difference between requirements and responsibilities. Since both are elements of the system in the problem space, they

might appear to describe the same system motivation. In the IEEE Software Engineering Book Of Knowledge (SWEBOK) [45], the requirement is defined as:

> *A software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem. The problem may be to automate part of a task of someone who will use the software, to support the business processes of the organization that has commissioned the software, to correct shortcomings of existing software, to control a device, and many more. The functioning of users, business processes, and devices is typically complex. By extension, therefore, the requirements on particular software are typically a complex combination of requirements from different people at different levels of an organization and from the environment in which the software will operate.*

The requirements are therefore a result of conflicting concerns from the software system's stakeholders and the software system's environment. The requirement is defined as a property. The USAP responsibility on the opposite is not the result of conflicting concerns. The USAP responsibility is constructed solely to fulfill the usability quality concern for a specific task and it has distinctive characteristics that differs it from a requirement. In short, these are:

- Context - The USAP responsibility is always defined for a specific task for the fulfillment of the usability quality of that task.
- Localization - The USAP responsibility is always localized to a particular portion(s) of the system.
- Functionality - The USAP responsibility always describes a particular behavior of the particular portion(s) of the system to which it is localized.

Additionally, the processes in which the artifacts are integrated differ. The requirement artifact is integrated in the process of collecting stakeholders' concerns and eliciting these. The USAP responsibility is part of an architectural design process coupled to the processing of a general Usability Supporting Architecture Pattern. The USAP responsibility is therefore not specific for a commissioned system and its characteristics are expressed in a general fashion to be adapted by any system.

USAP Activity and Task

During the work of identifying "Alarm & Event" USAP forces, a task analysis was done to identify the tasks of the "Alarm & Event" sub-system's users. From the task analysis the forces should be identified leading to the construction of usability supporting responsibilities.

In the article "Task Knowledge Structures: Psychological basis and integration into system design." [53], Johnson and Johnson describes the importance of task analysis to assist software designers to construct computer systems which people find useful and usable:

> *One way to approach this goal is to assume that knowing something about how users approach and carry out tasks will aid software designers when making design decisions which will ultimately affect computer system usefulness and usability. As a result task analysis has emerged as an important aid to early design in HCI.*

Task analysis according to Johnson and Johnson is an empirical method which can produce a complete and explicit model of tasks in the domain, and of how people carry out those tasks. Even if the USAP study did not do a complete task analysis according to how task analysis is described [53], it used a number of the proposed data collection techniques for task analysis. The techniques used to identify the tasks for the "Alarm & Event" scenario were:

- Direct observations of commissioners demonstrating the "Alarm & Event" parts of the systems.
- Interviews with: commissioners of the systems, "Alarm & Event" system architects, support responsible for the systems.
- Studies of: documents describing the usage of the "Alarm & Event" systems, "Alarm & Event" guidelines e.g. the "Engineering Equipment & Materials Users' Association" (EEMUA) publication no. 191: "Alarm Systems - A Guide to Design, Management and Procurement" [54].

The analysis of the collected data was done as:

1. Identify the roles and created work products (goals) of the "Alarm & Event" parts of the current systems (which would be consolidated into a product line system).

2. Perform a task analysis of the activities involved in creating the work products of “Alarm & Event”.

3. Identify the objects used in performing the actions, e.g. “Raised Alarm” and “Alarm & Event condition”.

4. Reason around which ones of the tasks require architectural support, e.g. “Author an Alarm & Event condition”, “Handle a raised Alarm”.

The result is a hierarchy of activities with sub-activities called tasks. The activity is the highest level in the hierarchy and the task is the second highest level.

The most important aspects of the tasks with requirement on architectural support are formulated as responsibilities. In [55], the “cancellation” USAP is presented as a modified version of the Model-View-Controller (MVC) pattern first defined by Beck et al. [36]. The MVC pattern was extended with new components, connectors and responsibilities to accommodate the “Cancel” requirements on usability support.

Using the experience from the MVC pattern for “Cancel”, the MVC-pattern was used to test if it also could be modified to host the responsibilities for tasks involving the work products: “Alarm & Event Condition”, “User Profile”, and “Environment Configuration”. The MVC-pattern did not decide the responsibilities. The task analysis was the base for constructing usability-supporting architecture responsibilities. If the constructed responsibilities would not have been possible to assign to a modified MVC-pattern, either another pattern-solution would have been chosen as a base or a new architectural sample solution constructed from scratch.

The responsibilities are formulated as ways in which the system architecture must support the usability quality of the task in order to make the task useful and easy to perform. At the time, this resulted in 79 responsibilities. To structure the responsibilities, they were classified according to the common activities they support. This resulted in a hierarchy of activities and tasks, and the tasks’s responsibilities. After a review of the CMU/SEI team and an “Alarm & Event” expert at ABB, some of the responsibilities could be consolidated or removed which resulted in a list of 43 “Alarm & Event” responsibilities.

Further analysis discovered that the responsibilities from the processing of the three work products had been categorized in a very similar fashion according to the activities they participated in. The activities were versions of: authoring, execution, logging, and authorization. Placeholders for the activities

were identified that were furnished with the work product or role. The discovery was a break-through since the activities are general and applicable to the processing of more work products by furnishing the placeholder with the work product or role. Each activity had a set of tasks attached to it. "Authoring" had e.g. the tasks "Create an [Alarm & Event Condition]" and "Modify an [Alarm & Event Condition]". The tasks also made use of the placeholder and furnished it e.g. with the work product [Alarm & Event Condition].

During the continued analysis, it was discovered that the responsibilities were nearly identical for each activity task no matter if the responsibilities had been created for the "Alarm & Event" scenario, the "Environment Configuration" scenario or the "User profile" scenario. The difference could be described by using the activity place holder and furnishing it with the work product or role.

The discovery reduced the total number of responsibilities for the scenarios from over hundred to 31, since the scenarios could share common activities, each consisting of tasks and the tasks' usability-supporting architectural responsibilities. The common activities, tasks and responsibilities each had a placeholder furnished by the scenario's work product or the scenario's role making the activity, task, and responsibility scenario-specific.

If the processing of the work products is supported by common responsibilities, then the solution space also can be common. The architectural solution supporting the "Authoring" activity can be shared by the processing of all three work products. The shared solution just has to make room for different interpretations of the general activities' placeholder. That is, the common solution has to be able to offer the user a way of e.g. authoring both a "Alarm & Event Condition" as well as an "Environment Configuration", but the mechanisms behind how authoring is supported by components and their behavior could be the same.

What was discovered was a way of offering the architects re-usable solutions, supporting common activities for the processing of more than one system-environment work product for more than one role. Responsibilities are usually presented as parameters tagged to components in an UML-diagram. In [3], the reason why UML sample solutions did not work for the industrial software system domain is explored. The idea surfaced of adding a responsibility implementation description to each responsibility description. For each responsibility, the portions of the system and their behavior, implementing the responsibility, are described.

The architects are offered one responsibility at a time together with a textual description of how this responsibility can be implemented by portions of

the system and the portions' behavior. It is not stated what the portions should or could be or what pattern the solution should be based on. In this way the architects can read the responsibility implementation description and visualize how the wording "portions of the system" might be translated into their own architectural design. If the architects feel that parts of the architectural design are in place to support the responsibility in the way the responsibility implementation describe, then they do not have to change the architecture in order to implement that specific responsibility.

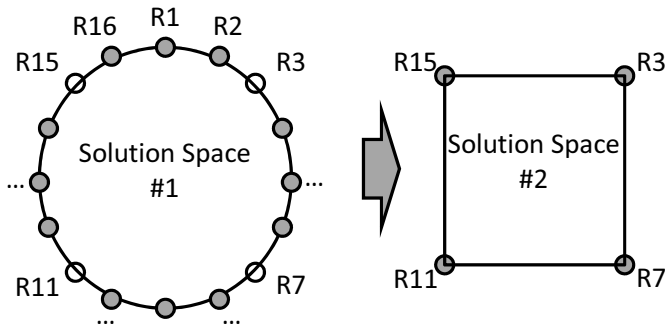

Figure 7: Examples of solution spaces spanned by two different sets of chosen points

This way of presenting responsibilities is like putting a magnifying glass over a very small part of a sample solution which lets the architects translate what they see from this very small part into their own design. Depending on what responsibilities the architects choose, the solution space will be different. This is illustrated with a set of points in a two-dimensional space, see Figure 7. Depending on what points are chosen the resulting space spanned by the points will take on different shapes. For a software architecture, it's not the shape that will look different but the set of components and their interactions implementing the chosen responsibilities.

In the "System-Environment Interaction Hierarchy" in Figure 8, the USAP work product processing considered in the USAP field study are "system-operational environment interaction" work products. As previously discussed in the Section F.3, software quality concerns not observable in runtime as e.g. maintainability would be concerns of processing of "system-development environment interaction" work products. The "System-Environment Interaction

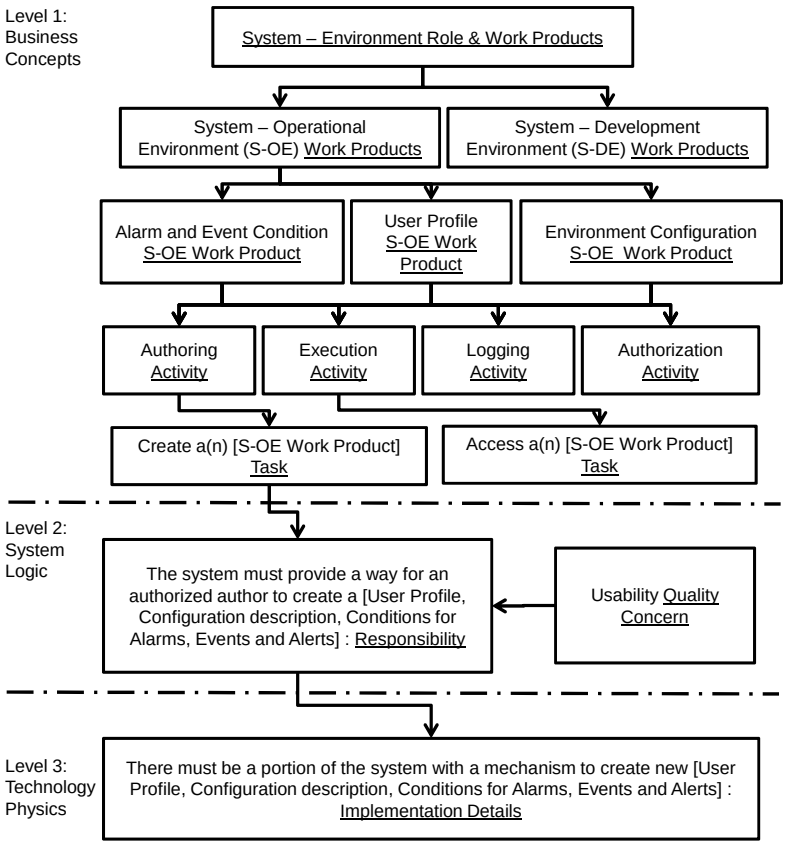

Figure 8: System-Environment Interaction Hierarchy with three levels

Hierarchy" has three levels: Business Concepts perspective; System Logic perspective, and Technology Physics perspective. The task analysis done in the USAP field study studied the interactions between the system and its operational environment. For the interactions between the system and its development environment, the task analysis has to study how architects, developers, project managers etc work with the development of the system. A task analysis of the development environment would result in the processing of "system-development environment interaction" related work products with us-

ability concerns from the development environment. The system-environment interface, in that case, would be the test/build/implement system-development environment interfaces.

At the time of the execution of the field study, the family of activities, tasks, responsibility descriptions and responsibility implementation descriptions were called a “Foundational Pattern” to align the USAP with the spirit and work of Alexander [56][57]. The idea of a “Foundational Pattern” is described in more detail by John et al. [58].

F.5.2 Classification of USAP artifacts

The extracted artifacts from the USAP concept are:

System Environment Business Roles and Work Products - describes the system environment’s roles and work products.

System Environment Interface - describes system’s environment interface, e.g. customer UI or development environment (build/test/implement) UI.

Quality attribute - describes a feature or characteristic that affects an item’s quality according to IEEE 610 [44].

System-Environment Interaction Scenario - describes an interaction between the system and its roles, e.g. a use case or a quality attribute scenario.

Activity - describes an activity involved in the System Environment Business Roles’ creation of Work Products.

Placeholder - describes the role or work product. Is used by the activity, the activity’s tasks and their responsibilities, in order to make them specific to the work product or the role.

Task - describes a task of the activity.

Responsibility Description - describes how the system must interact with its environment to ensure that a specific quality attribute concern of the task is met.

Responsibility Implementation - describes the implementation of the responsibility as particular portion or portions of the system and their behavior.

Pattern Responsibility Description - describes a responsibility of an established pattern from e.g. [59][37].

Pattern Responsibility Implementation - describes the implementation of the responsibility as components and connectors [59][37].

Rules & Guidelines - describes existing quality-specific, domain specific, rules & guidelines for how the system should interact with its environment in order to have a certain quality.

Note that if the system environment interface is a build, test, or implementation interface between the system and its developers than the roles and work products are the development's roles and work products. The system-role interaction scenario will then describe how the tester or builder interact with the system. In this case the site-dimension of the Software Engineering Taxonomy is the software development organization's site. If the system environment interface is the interface between the system and its customers/users, then the roles and work products are the customer's roles and work products. For the last case, the site dimension of the Software Engineering Taxonomy is the customer's. Figure 9 shows the classification of the USAP artifacts into the Software Engineering Taxonomy.

F.5.3 USAP Information Description-Selection Process

This section describes the flow of describing or selecting the USAP information. The flow uses the classified artifacts in the Software Engineering Taxonomy, Figure 9, and describes a sequence that follows Zachman's consistency rules and uses the experience from how the "Alarm & Event" USAP was created. The result is the USAP Information Selection/ Description Process, which is visualized in Figure 10.

Notice that no step changes both the usage perspective and the information abstraction to align with Zachman's fifth rule of excluding diagonal steps in the framework when constructing process composites. There are two start alternatives: Existing system-environment interface, or the system-environment domain's Business Roles & Work products. The first option presumes that a system environment interface is at hand, e.g sketch or legacy UI. For the product line system in the field study, the start was the legacy user interfaces of the systems to be part of the product line. The legacy user interfaces are then described/ selected. Then follows a description/selection of reusable

Abstraction → / System Environment Perspective ↓	Process Transformations (HOW)	Organization Groups (WHO)	Motivation Reasons (WHY)
Business Concepts	• Activity • Task • Placeholder	• Business Roles & Work Products	
System Logic	• Responsibility Description • Pattern Responsibility Description	• System-Environment Interaction Scenario	• Quality Attribute • Rules & Guidelines
Technology Physics	• Responsibility Implementation Description • Pattern Responsibility Implementation Description	• System-Environment Interface	

Figure 9: USAP artifacts classified in the Software Engineering Taxonomy. The environment can either be the system's operational environment or the system's development environment

system-environment interaction scenarios, with requirements on usability support in the architecture not solved by separating the system-environment interface logic from the rest of the system's logic. Reusable system-environment interaction scenarios can be chosen from the scenario listing of Bass and John [60] or the Usability Patterns from Juristo et al. [61][62][63]. The USAP field study used the USAP scenarios: "System Feedback" and "User Profile" [60][64]. The latter was divided into "User Profile" and "Environment Configuration".

If the start would have been the system-environment domain's Business Roles & Work products, then the reusable system-environment interaction scenarios are described/ selected in parallel with the description/ selection of system-environment domain's Business Roles & Work products. For example, a large set of roles and work products are at hand. By using the reusable system-environment interaction scenarios, the roles and work products related to the scenario can be identified. These roles and work products need usability support in the architecture for the system's implementation of their activities and

tasks. The description of the system-environment domain's Business Roles & Work products must be the step before describing/ selecting reusable activities and tasks. Otherwise the furnishing parameter of the activity placeholder can not be identified.

The roles and work products are described/selected in the next step. In the USAP study the roles were: system commissioner and system operator. The work products were: "Alarm & Event Condition", "User Profile" and "Environment Configuration". By describing/selecting multiple work products, the general activities involved in the processing of the role's work product can be identified.

When the tasks are described/ selected, the task's placeholder is furnished by the description/ selection of role or work product. In the USAP field study the place holder was furnished with the "Alarm & Event condition", "Environment Configuration", and "User Profile" for the majority of the tasks. For the authorization tasks, the placeholder was furnished with the role, "Author" and "User". The role or work product, furnishing the placeholder is used by the activity, responsibility description, and the responsibility implementation.

Responsibilities are described/ selected, using: quality attribute information, rules & guidelines information, pattern responsibilities, scenario information and task information. The USAP responsibility used the usability quality attribute and hence supports usability in the architecture.

The final step, the description/selection of the responsibility implementation, is the view with the Technology Physics perspective and the Process Transformation abstraction, since in this view architects describe components and connectors. By examining the responsibilities' implementation from this view, the architects can compare the responsibility implementation description with their design, without changing their mind-set to another information abstraction and usage perspective. The description/selection of the responsibility implementation uses information from: the responsibility description and possibly, existing pattern responsibility implementation descriptions.

If architects immediately would view architecture patterns, which have the Technology Physics perspective and the Process Transformation abstraction, after considering requirements or user's roles and work products, the diagonal step in the Software Engineering Taxonomy would introduce inconsistencies in the descriptions and a difficult shift in mind-set between both information abstractions and usage perspectives. Using the Software Engineering Taxonomy for classifying elements of the USAP and for incorporating the elements in the USAP information description/selection process, contributes to a harmonized sequence of process steps with a end-product that matches the expectations of

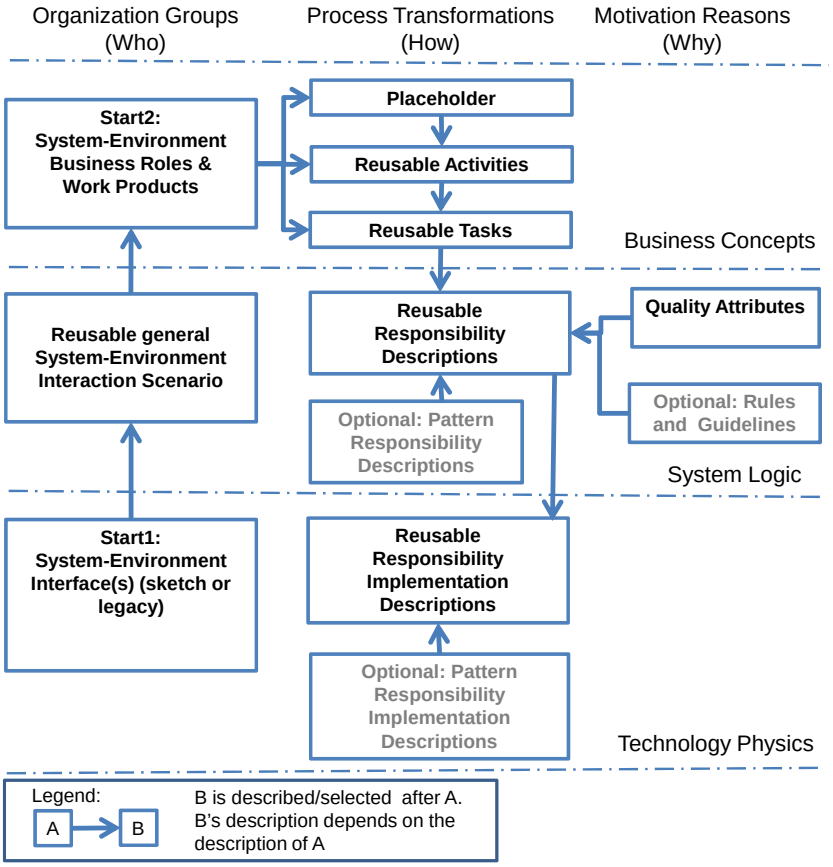

Figure 10: USAP information description/selection process, using the classified artifacts from the Software Engineering Taxonomy. The figure describes in what order the USAP artifacts should be described or selected, guided by the USAP artifacts' classification view's location in the taxonomy.

the USAP information user.

The USAP field study included the design and implementation of the USAP information selection tool, presented in Paper **D**. The tool guided the architects through the USAP information description/selection process but offered only selection features.

F.5.4 Summary

The USAP artifacts were identified and classified in the Software Engineering taxonomy. The classification of the USAP artifacts showed how the artifacts can be arranged in a process composite to describe the USAP information description/ selection process. Some new discoveries were made during the analysis of the classified artifacts:

- The inclusion of a traditional enterprise perspective, the business concepts perspective, led to discoveries of new interrelationships between the USAP artifacts: system-environment interaction scenario, system environment business roles & work products, system-environment activities and tasks related to the roles & work products, responsibility descriptions, quality attributes, and responsibility implementation descriptions.

- System environment business roles and work products are a key artifact in linking the USAP scenario to common activities and tasks supporting more than one role or more than one work product.

- System environment may be operational or development environment. The environment decides what system-environment interface, business roles and work products should be used in the USAP information description/ selection process.

- The placeholder of the common activity is furnished by the work product or the role.

- The responsibility is related to the quality chosen to be supported for the scenario. For USAP, the usability quality is supported by the USAP scenarios. Possibly, the USAP information description/ selection process can be used for other quality scenarios, if their tasks' quality concern can be expressed as responsibilities.

F.6 Conclusions and Future Work

The Software Engineering taxonomy can serve as a reasoning framework into which artifacts of software engineering case and field studies can be classified for the creation of process composites or for further analysis. For the Influencing Factors method and the Sustainable Systems Case study, the data was classified and analyzed. For the USAP field study, the data was classified and used for process composite creation. Applying the Software Engineering Taxonomy led to the additional contributions:

- Sustainable systems case study
  - The sustainable key-competences in the industrial software system development organization carry the application domain knowledge and the system knowledge, thereby increasing the social sustainability of the company. The sustainable key-competences pass the knowledge on to the system developers during informal design discussions.
  - The development organizations sustain economical capital by planning for changes when the changes are technology changes. When the changes are organizational, e.g. distributed development, the management have lost social capital by failing to plan for how the development organization has to adapt to the new work-form. It has been too little known in the companies, what requirements a distributed development environment has on the development organization's structures and communication.
  - The incorporation of a remotely located development team in the development organization will be especially difficult in a culture that has social capital invested in sustainable key-competences and their informal spreading of knowledge. If the organization has ignored investigating in explicit software documentation, increasing the tangible economical capital, the new remotely located team can make use of neither the social capital nor the economical capital related to system know-how.
  - The sustainable target market increases the intangible economical capital.
  - Intangible economical capital in the form of goodwill and reputation is increased by delivering reliable systems for a long-time to the target markets.

- The propositions regarding the importance of intangible economical capital of explicit defined roles and hand-over of information along with explicit business goals communicated to the entire organization were rejected in the case study.
- The social capital in the form of implicit roles, well-known to the developers, is replacing the economic capital in the form of formal descriptions of roles and formal communication.
- The case study's propositions regarding the importance of control of the the cost, quality, and schedule for sustainable development remain to investigate. The investigation has to include interviews with project leaders and line management. The case study assumed that the product managers, software architects, and senior developers would contribute to the control of cost, quality and schedule. This turned out to be a false assumption. The product managers, software architects, and senior developers had little or no insights into how Key Performance Indicators were measured or how schedule control was exercised.
- The list of success-critical concerns for sustainable development does not include as many architectural success-critical concerns as expected. This could be related to the lack of consensus around the concept of software architecture. The lack of a consistent software architecture definition and tools and methods based on such definition might make the industry reluctant to embrace the concept of software architecture. Risks are not welcome in industrial software systems that have to live for decades. The business case arguing added value of software architecture for sustainable development is simply not good enough for the three investigated cases in the domain of industrial software systems.
- In order to increase tangible economical capital in the form of software engineering process artifacts, e.g. architecture descriptions, the companies must first increase the tangible economical capital in form of organizational artifacts, e.g. role descriptions and social capital in form of information communication channels. Curtis's study [20][21], the Dikel study [33] and the Sustainable Industrial Software Systems case study point toward a conclusion that sustainable development concerns related to the software development organization, must be addressed first before software engineering tools and methods could have a significant impact on sustainable

development.

- Influencing Factors field study
  - Additional observations regarding stakeholder role and stakeholder perspective. For the stakeholders with the Business Concepts perspective, maintainability and testability are discussed among stakeholders as software development improvement strategies, e.g. distributed development or introduction of product lines. The architectural structures for realizing these strategies are seldom discussed among the success-critical stakeholders. Decisions regarding architectural structures are taken informally by the architects. This is a noticeable difference between the software engineering discipline and the building engineering discipline, where building structures are discussed by architects, customers, and contractors.
- USAP field study
  - The inclusion of a traditional enterprise perspective, the business concepts perspective, led to discoveries of new interrelationships between the USAP artifacts: system-environment interaction scenario, system environment business roles & work products, system-environment activities and tasks related to the roles & work products, responsibility descriptions, quality attributes, and responsibility implementation descriptions.
  - System environment business roles and work products are a key artifact in linking the USAP scenario [64] to common activities and tasks supporting more than one role or more than one work product.
  - System environment may be operational or development environment. The environment decides what system-environment interface and business roles and work products should be used in the USAP information description/ selection process .
  - The placeholder of the common activity is furnished by the work product or the role.
  - The responsibility is related to the quality chosen to be supported for the scenario. For USAP, the usability quality is supported by the USAP responsibility. Possibly, the USAP information description/ selection process can be used for other quality scenarios, if their tasks' quality concern can be expressed as responsibilities.

When classifying artifacts, not all of the 30 cell descriptions in the taxonomy need to be used. The Influencing Factors analysis used three cells, the USAP analysis used six cells. The Sustainable Industrial Software System case study used 19 cells showing that sustainability is a concept with a large set of descriptions and interactions between the descriptions.

It remains to implement the description features in the USAP information description/selection tool. This is done in an ongoing research project. If it is the case that the placeholder always can be furnished with either role or work product or not remains to validate by describing additional USAPs. Possibly, the USAP information description/ selection process can be used for other quality scenarios, if their tasks' quality concern can be expressed as responsibilities.

For the Sustainable System study, it remains to use the classification of sustainable development concerns for set-up of goals and metrics in order to address some of the concerns the companies felt they could meet in a better way. The interrelationships between the classified concerns could then be used to create a process, in the same manner as the USAP information description/ selection process was created.

Bibliography

[1] P. Stoll, A. Wall, and C. Norström. Software Engineering featuring the Zachman Taxonomy. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-240/2009-1-SE, Mälardalen University, School of Innovation, Design and Engineering, 2009.

[2] P. Stoll, L. Bass, B. E. John, and E. Golden. Preparing Usability Supporting Architectural Patterns for Industrial Use. Proceedings of International Workshop on the Interplay between Usability Evaluation and Software Development (I-ISED), Pisa, Italy, 2008.

[3] P. Stoll, L. Bass, B.E. John, and E. Golden. Supporting Usability in Product Line Architectures. Proceedings of the 13th International Software Product Line Conference (SPLC), San Francisco, USA, August 2009.

[4] P. Stoll, A. Wall, and C. Norström. Guiding Architectural Decisions with the Influencing Factors Method. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008, 2008.

[5] P. Stoll and A. Wall. Business Sustainability for Software Systems. Proceedings of Business Sustainability, Ofir, Portugal, 2008.

[6] J. F. Sowa and J. A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM System Journal*, 31:590–616, 1992.

[7] J. A. Zachman. A Framework for Information Systems Architecture. *IBM Systems Journal*, 26(3):276–292, 1987.

[8] J. A. Zachman. *The Zachman Framework for Enterprise Architecture; A Primer for Enterprise Engineering and Manufacturing*. Zachman International, 2003.

[9] J. A. Zachman. The Zachman Framework and Observations on Methodologies. *Business Rules Journal*, 5(11), 2004.

[10] P. B. Kruchten. The "4+1" View Model of architecture. *Software, IEEE*, 12(6):42–50, Nov 1995.

[11] R. Hilliard. Systems and software engineering - Recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1–24, 15 2007.

[12] ISO/IEC 10746 - 3: 1996, Information technology - Open distributed processing - Reference model: Architecture, 1996.

[13] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, second edition, 2003.

[14] C. O'Rourke, N. Fishman, and W. Selkow. Enterprise Architecture, Using the Zachman Framework. *Thomson Course Technology*, 2003.

[15] P. Pollan. Our decrepit food factories. *New York Times*, 2007.

[16] G.C Unruh. Escaping carbon lock-in. *Energy Policy*, vol. 30(no.4):pp. 317–325, 2002.

[17] G.H. Brundtland. Our common future. Report of the World Commission on Environment and Development. Published as Annex to General Assembly document A/42/427, 1987.

[18] T. Dyllick and K. Hockerts. Beyond the business case for corporate sustainabilityt. *Business Strategy and the Environment*, 11:130–141, 2002.

[19] R. K. Yin. *Case study research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. SAGE Publications, third edition, 2003.

[20] W. Curtis, H. Krasner, V. Shen, and N. Iscoe. On building software process models under the lamppost. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 96–103, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[21] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. Communications of the ACM, Vol. 31 No. 11, pp. 1268-87., 1988.

[22] B. Randell. The 1968/69 NATO Software Engineering Reports. Available at: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html [Accessed 20. June 2009], 1996.

[23] E. Dijkstra. The structure of the "THE"-multiprogramming system. *Commun. ACM 11*, 5:341–346, 1968.

[24] V. R. Basili and J. D. Musa. The future engineering of software: A management perspective. *Computer*, 24(9):90–96, 1991.

[25] M. Jackson. Will there ever be software engineering? *IEEE Software*, pages 36–39, 1998.

[26] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[27] C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm. On the definition of software system architecture. In *ICSE 17 Software Architecture Workshop*, 1995.

[28] P. Johnsson. *Enterprise Software System Integration: An Architectural Perspective*. PhD thesis, Industrial Information and Control Systems, Royal Institute of Technology (KTH), Stockholm, Sweden, 2002.

[29] R. Malveau and T. J. Mowbray. *Software Architect Bootcamp*. Prentice Hall Professional Technical Reference, 2003.

[30] J. O. Coplien. Borland software craftsmanship: A new look at process, quality and productivity. In *5 th Annual Borland International Conference*, 1994.

[31] M. E. Conway. How do committees invent? *Datamation magazine*, 1968.

[32] B. G. Cain, J. O. Coplien, and N. B. Harrison. Social patterns in productive software development organizations. *Annals of Software Engineering*, 1996.

[33] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying software product-line architecture. *Computer*, 30(8):49–55, Aug 1997.

[34] W. R. Ashby. *An Introduction to Cybernetics*. First Edition, Chapman and Hall: London, UK, 1956.

[35] D. Kane, D. Dikel, and J. Wilson. *Software Architecture: Organizational Principles and Patterns.* Prentice Hall, 2001.

[36] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. Technical Report Technical Report No. CR-87-43, Apple Computer, Inc. and Tektronix, Inc., 1987. Submitted to the OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming.

[37] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture A System of Patterns*, volume 1. Wiley, first edition, 1996.

[38] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 406–431, London, UK, 1993. Springer-Verlag.

[39] M. Fowler. *Pattern Of Enterprise Application Architecture*. Addison-Wesley, 2003.

[40] J. O. Coplien. Organization and architecture. 1999 CHOOSE Forum on Object-oriented Software Architecture, 1999.

[41] B. Boehm, Abts C., A. Winsor Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[42] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[43] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[44] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages –, Dec 1990.

[45] P. Bourque and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[46] Z. Antolic. An Example of Using Key Performance Indicators for Software Development Process Efficiency Evaluation. Technical Report, R&D Center, Ericsson Nikola Tesla d.d., 2008.

[47] J. O. Coplien. *Multi-Paradigm Dedign for C++*. Addison-Wesley, Reading, MA, 1998.

[48] N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives*. Addison-Wesley, 2005.

[49] I. Jacobson, M. Griss, and P. Jonsson. Making the reuse business work. *Computer*, 30(10):36–42, Oct 1997.

[50] Ilan Oshri, Julia Kotlarsky, and Leslie P. Willcocks. Global software development: Exploring socialization and face-to-face meetings in distributed strategic projects. *The Journal of Strategic Information Systems*, 16(1):25 – 49, 2007.

[51] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. *ACM SIGPLAN Notices*, 24(10):1–6, 1989.

[52] R. Wirfs-Brock and A. McKean. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley, 2003.

[53] H. Johnson and P. Johnson. Task Knowledge Structures: Psychological basis and integration into system design. *Acta Psychologica*, 78:3–26, 1991.

[54] EEMUA. 191 Alarm Systems - A Guide to Design, Management and Procurement . *Available: http://www.eemua.co.uk*, 2007, 2nd edition, ISBN 0 85931 155 4.

[55] E. Golden, B. E. John, and L. Bass. The value of a usability-supporting architectural pattern in software architecture design: A controlled experiment. In *Proceedings of the 27th International Conference on Software Engineering, ICSE*, St. Louis, Missouri, May 2005.

[56] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[57] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA, 1977.

[58] B. E. John, L. Bass, E. Golden, and P. Stoll. A responsibility-based pattern language for usability-supporting architectural patterns. Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS), Pittsburgh, PA, US, 2009.

[59] E. Gamma, R Helm, R. Johnson, and J. Wissides. *Design Patterns - Elements of Reusable Object-Oriented Sojlware*. Addison-Wesley, 1995.

[60] L. Bass and B. E. John. Linking usability to software architecture patterns through general scenarios. *The Journal of Systems and Software*, 66:187–197, 2003.

[61] N. Juristo, H. Windl, and L. Constantine. Introducing usability. *Software, IEEE*, 18(1):20–21, Jan/Feb 2001.

[62] N. Juristo, M. Lopez, A. Moreno, and M.-I. Sanchez-Segura. Improving software usability through architectural patterns. Paper presented at the ICSE 2003 Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction, Portland, Oregon, USA., 2003.

[63] N. Juristo, A.M. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *Software Engineering, IEEE Transactions on*, 33(11):744–758, Nov. 2007.

[64] L. Bass, B. E. John, and J. Kates. Achieving usability through software architecture. Technical Report No. SEI-TR-2001-005, Carnegie Mellon University/Software Engineering Institute, Pittsburgh, PA, 2001.