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

September 24, 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 to make the general activity and tasks 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 composed by following Zachman's consistency rules in the Software Engineering Taxonomy.

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 [1], a derivative of the Zachman framework called the Software Engineering Taxonomy is suggested for the classification of software engineering information. 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].

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 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.*

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

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:

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, the view, in the Zachman framework can be aligned with the ISO/IEC 42010:2007 viewpoints according to the ISO/IEC 42010:2007 version [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,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.

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.

The Software Engineering Taxonomy is described further in [1].

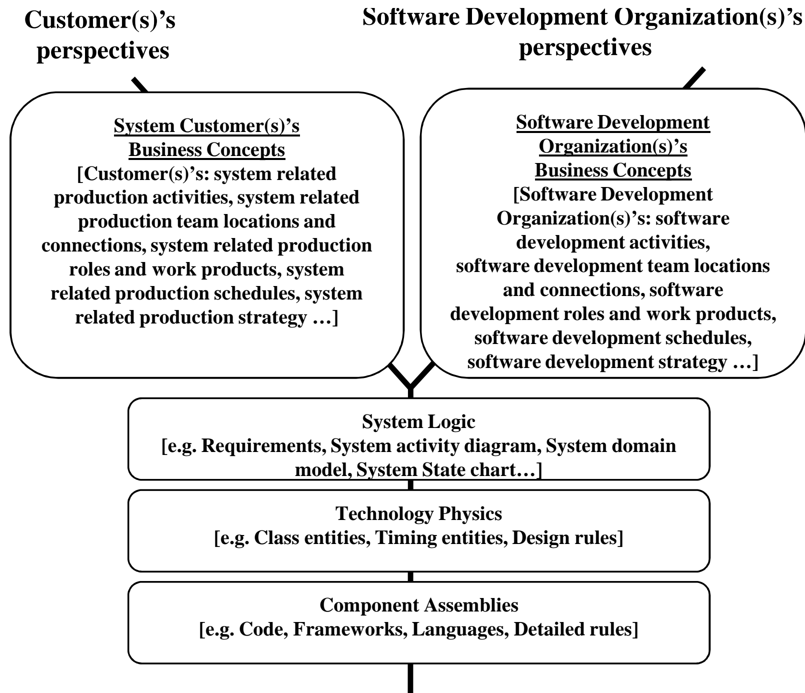

Figure. 2: The Customer's and the Software Development Organization's perspectives

3 Software Engineering Taxonomy and System Sustainability

The sustainable industrial software systems theory presented in paper [5] 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 theory in [5] 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.

3.1 Sustainable Industrial Software System Development

Pollan [15] 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". Unruh [16] has argued that numerous barriers to sustainability arise because today's technological systems were designed and built for permanence and reliability, not change.

"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.*

Dyllick and Hockerts [18] 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.

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 despite changes in concerns originating from: new technology, new stakeholder needs, new organizations,

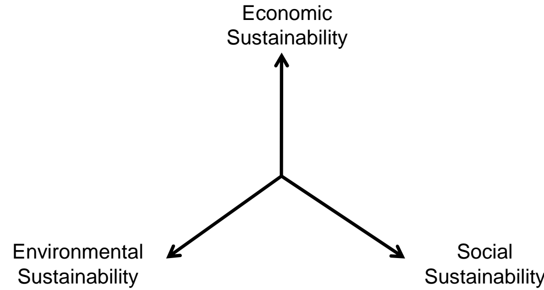

Figure. 3: Three dimensions of corporate sustainability

and new business goals during decades is a true challenge. 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 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.

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 [5] 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 generalizes 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 which also develops large complex long-lived software systems.

Reliability: Structured individual interviews were conducted which were approximately three hours long on 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 send 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 system will have an organization with a high communication interaction
6. We believe that organizations that manage sustainable systems will have an organization with clear 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 answers to the 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.

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.

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.

It's evident that even if the term software engineering was coined in the 1968 NATO Software Engineering Conference[1] and Dijkstra described software structures the same year [22], the usage of software engineering and software architecture concepts and tools in the domain of industrial software systems is low.

Basili and Musa write in an article 1991, 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 in [23]. 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,

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

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

by definition, be a professor of unsolved problems" [24]. 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 [25], 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*

Bass, Clements, and Kazman [13] 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 [26], 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 in his PhD thesis [27] 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 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 in [24].

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' reasoning. In [21], 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.

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 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.

Malveau and Mowray [28] suggests 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 as is suggested in the theory in [5].

In the Pasteur research project at Bell Labs [29], Coplien et al. investigated organizational structures. Coplien's organizational studies found two organizational patterns:

- Architecture Follows Organization, a restatement of Conway's Law [30].
- 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, Coplien and Harrison have described additional organization patterns in [31]. Their conclusion is that: "If there is one consistent measure of successful organization, it is how well its members maintain relationships through communication."

Dikel, Kane, Ornburn, Loftus, and Wilson developed organizational principles in an effort to predict the success or failure of software architectures for large telecommunications systems [32]. In the case study, reported on in [32], 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 W.R. Ashby [33], the *law of requisite variety*[2], 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 [34], Kane, Dikel and Wilson describe 30 organizational patterns and 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 [28] patterns (discussed by Beck [35], Buschman [36] , Shaw [25], Gamma [37] and Fowler [38]) as well as the macro design-level [28] patterns (discussed by Fowler [38], Coplien [39] and Kane [34]) 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. [40] describes MBASE that considers architectural complexity.; Halstead [41] proposes measures to predict understanding effort based on grammatical complexity of code modules. McCabe [42] 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 are marked with: *** for very high importance, ** for high importance, and * for importance. 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***

[2] http://pespmc1.vub.ac.be/REQVAR.html

(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 for 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.

In [20] 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. 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 KPI 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 [43], adopted by the IEEE Software Engineering Book Of Knowledge (SWEBOK) [44], 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 [45]: 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: Boehm et al. [40], Halstead [41] or McCabe [42].

The customer-specific architectural change projects was a sustainability concern voiced by all developers and architects. This confirms the top-two finding in the Curtis study [21] related to fluctuating requirements as a hinder for software development productivity. One architect in the Curtis 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 [46] as one way of finding commonalities for a system's target market. 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 by 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 custome rlocation 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 [2][3] of the interplay between usability and software architecture, isolating the user interface logic is not enough to achieve a usable system. 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. The IEEE Software Engineering Body Of Knowledge (SWEBOK) published 2004 [44], mentions the word usability six times but refer to the software ergonomics discipline for how to work with usability. Rozanski and Woods only suggest the isolation of user interface logic as a usability tactic, in contrast to their thorough descriptions of ten security tactics in their book "Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives" published 2005 [47].

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. Jacobson et al. discusses the reuse of components in [48] 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 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.

The non-balance of short-term needs and long-term needs when setting business goals has been described by Dyllick and Hockerts in their article on Corporate Sustainability [18] as:

> *In recent years, driven by the stock market, firms have tended to overemphasise 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.*

3.4.1 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 and as such minimizes the risk for new customers speculating in what system to buy.
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 how to work distributed. 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 not 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.

3.4.2 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 system 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

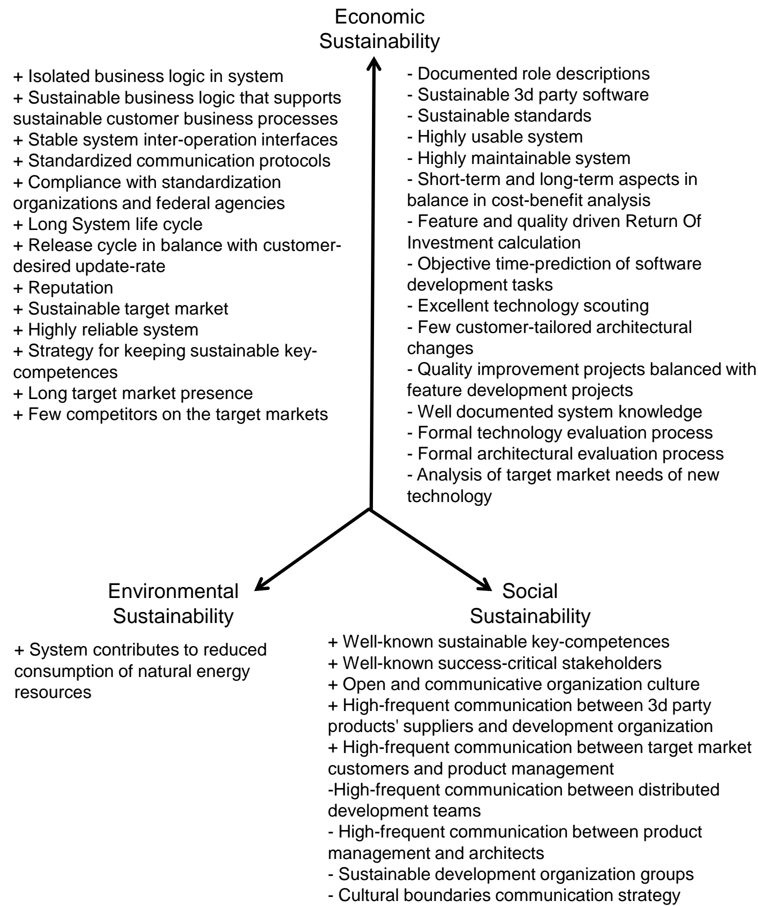

Figure. 5: Three dimensions of important sustainable development capital in the domain of Industrial Software System according to the findings. The “plus” sign indicates that the companies felt they had the capital. The “minus” sign indicates they felt they needed an improvement.

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 in 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. argues that, in order to achieve successful collaboration, firms should consider investing in the development of socialization despite the constraints imposed by global distribution [49]. The socialization efforts could be e.g. increased communication through virtual Face to Face (F2F) meetings, kick-of meeting, progress meetings etc.

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 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 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 study[20] [21], Kane's study[34], 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.

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. The way the Influencing Factor is used 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 the matrix used in [4], the IF 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.

In [4], it is described 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 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 described in [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, presented in [4], are here used for additional investigation using the Software Engineering Taxonomy [1] 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.

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 ownerships 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 of the development organization if the customer had not voiced the goal or concern related to the goal.

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

Abstraction →	MOTIVATION REASONS (WHY)		
Software Development Organization Perspective ↓			
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.

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 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 current calculations much favored the short-term gains.

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 architectural 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.

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.

5.1 USAP Artifact Identification

5.1.1 USAP Responsibility

Even though the word "responsibility" has been used in the publications of USAP [2][3], it has never got it's own definition in the context of a 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 Ward Cunningham and Kent Beck [50] . 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 [50])

The responsibility as described by Beck and Cunningham was later used by Frank Buschmann et al. in the book "Pattern-Oriented Software Architecture A system of Patterns" [36] to describe the responsibilities of classes in architectural patterns. 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 [36])

Wirfs-Brock uses responsibilities in the book "Object Design: Roles, Responsibilities, and Collaborations object-oriented design" [51] in the same sense as Beck and Cunningham. She defines the responsibility as:

A responsibility = an obligation to perform a task or know information

(p. 3 [51])

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. The requirement is defined by the IEEE Software Engineering Book Of Knowledge (SWEBOK) [44] 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 or portions 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.

5.1.2 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." [52], 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 in [52], 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[3]: "Alarm Systems - A Guide to Design, Management and Procurement"

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"

[3]http://www.eemua.co.uk/p_instrumentation.htm

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. The "cancellation" USAP [53] is a modified version of the Model-View-Controller (MVC) pattern first defined by Beck et al. in 1987 [35]. 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 basis 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 basis or a new architecutral 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 amount 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.

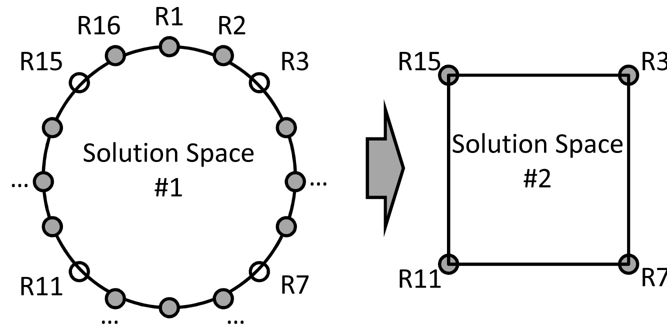

Figure. 7: Examples of solution spaces spanned by two different sets of chosen points

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] it is explained why UML sample solutions did not work for the industrial software system domain. 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.

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 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

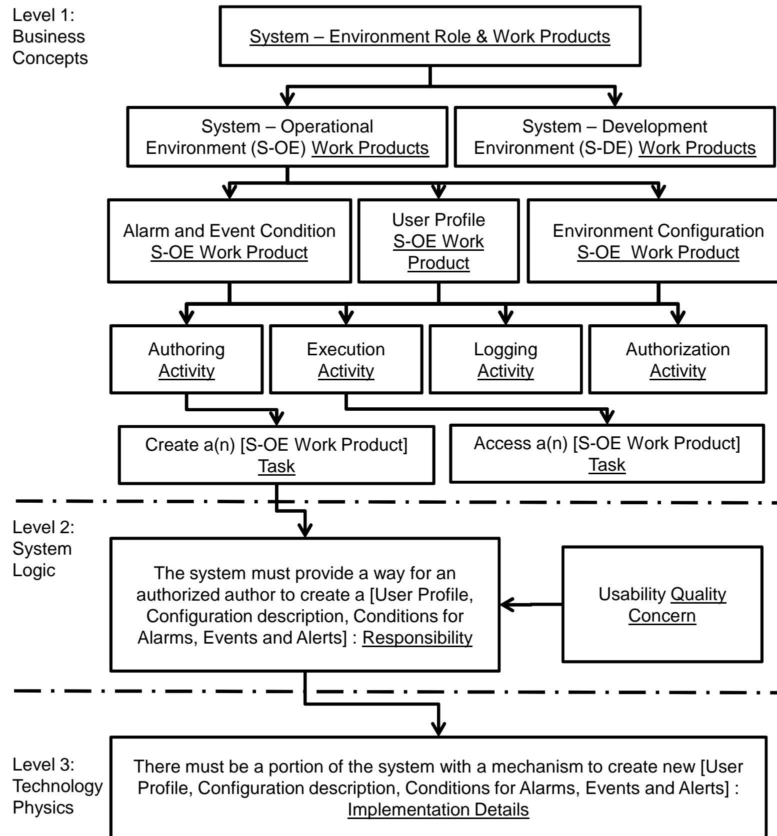

Figure. 8: System-Environment Interaction Hierarchy with three levels

"System-Environment Interaction 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 usability 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 [54] [55]. The idea of a "Foundational Pattern" is described in more detail by John et al. in [56].

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 [43].

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. [57] [36].

Pattern Responsibility Implementation - describes the implementation of the responsibility as components and connectors [57] [36].

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.

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

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

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 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[58] or the Usability Patterns from Juristo [59] [60] [61]. The USAP field study used the USAP scenarios: "System Feedback" and "User Profile" [58][62]. 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 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 identi-

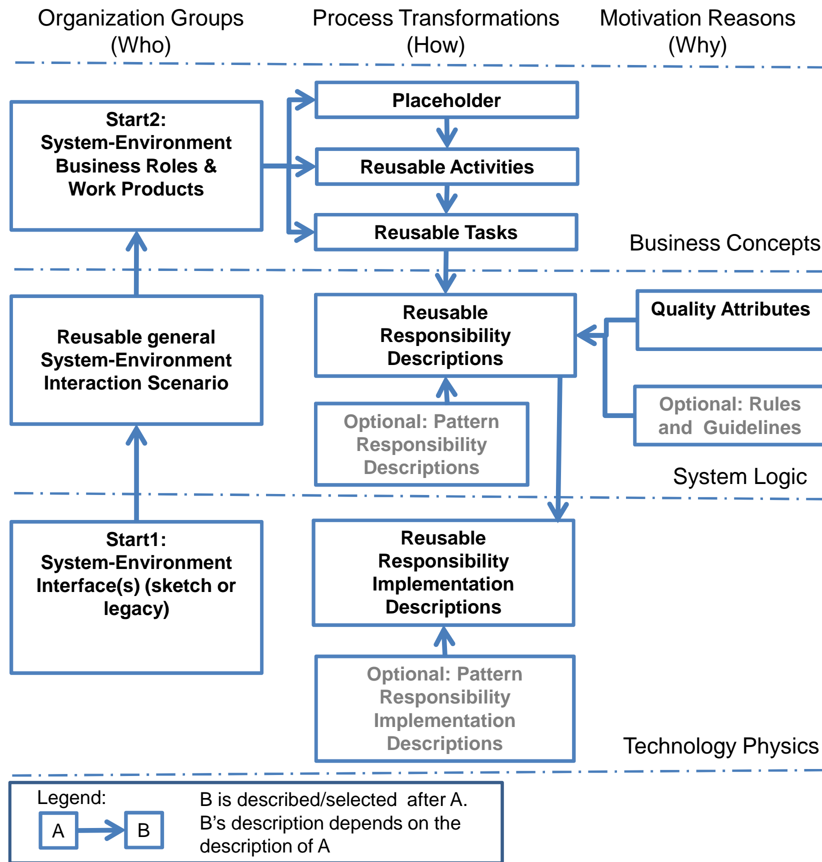

Figure. 10: USAP information description/selection process, using the classified artifacts from the Software Engineering Taxonomy. The figure describes in what order the USAP artifacts should be described or selected, guided by the USAP artifacts' classification view's location in the taxonomy.

fied.

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 infor-

mation. 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 viewing the responsibilities' implementation from this view, the architects can compare the responsibility implementation description with their design view, 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 the USAP information description/selection process, contributes to a harmonized sequence of process steps with a end-product that matches the expectations of the USAP information user.

The USAP field study included the design and implementation of the USAP information selection tool, presented in [3]. The tool guided the architects through the USAP information description/selection process but offered only selection features.

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 [62] 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 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.

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-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 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 have 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 system 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 study [20] [21], the Dikel study [32] 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 [62] 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 nearly all 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 the placeholder always can be furnished with either role or work product 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.

References

[1] P. Stoll, A. Wall, and C. Norström. Software engineering featuring the zachman taxonomy. Technical report, Mlardalen 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] E. Dijkstra. The structure of the “THE”-multiprogramming system. *Commun. ACM 11*, 5:341–346, 1968.

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

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

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

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

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

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

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

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

[31] B. G. Cain, J. O. Coplien, and N. B. Harrison. Social patterns in productive software development organizations. *Annals of Software Engineering*, 1996.

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

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

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

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

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

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

[43] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages –, Dec 1990.

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

[45] 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.

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

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

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

[49] 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.

[50] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. *ACM SIGPLAN Notices*, 24(10):1–6, 1989.

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

[52] H. Johnson and P. Johnson. Task Knowledge Structures: Psychological basis and integration into system design. *Acta Psychologica*, 78:3–26, 1991.

[53] 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.

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

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

[56] 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.

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

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.