TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
*Julkaisu 843 • Publication 843*

Sari Vesiluoma

# Understanding and Supporting Knowledge Sharing in Software Engineering

Tampere 2009

Tampereen teknillinen yliopisto. Julkaisu 843
Tampere University of Technology. Publication 843

Sari Vesiluoma

# Understanding and Supporting Knowledge Sharing in Software Engineering

Thesis for the degree of Doctor of Philosophy to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 27th of November 2009, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2009

ISBN 978-952-15-2250-5 (printed)
ISBN 978-952-15-2263-5 (PDF)
ISSN 1459-2045

# ABSTRACT

Very often in practice, the problems in software engineering projects seem to have their origins in knowledge sharing difficulties. This creates a need to understand knowledge sharing in the software engineering field better and to find ways to support it. Through an improved understanding of knowledge sharing and especially the problems in it, the targets for support can be determined. By supporting knowledge sharing the maturity of software development processes can be significantly raised.

This thesis presents new approaches and offers tools to understand and support knowledge sharing. The main results are the Knowledge Sharing Framework (KSF) as a tool for understanding and the Knowledge Sharing Pattern Language as a tool for supporting knowledge sharing.

The purpose of the KSF is to help to identify the current knowledge sharing situation in an organization. This creates the basis for improvement actions in an organization. In this thesis the KSF is utilized in an empirical case study as well as a being framework for understanding differences in knowledge sharing approaches in different software development methods. KSF profiles are used to describe the results of KSF analyses in an illustrative way.

After gaining more understanding through defining and using the KSF, the focus is transferred to how to support knowledge sharing. A technique, knowledge sharing patterns, is proposed here to improve existing software development processes with better knowledge sharing properties. Knowledge sharing patterns describe solutions to problems frequently related to the difficulties in knowledge sharing and offer a light-weight mechanism to introduce knowledge sharing sensitive practices in development processes. The knowledge sharing patterns form the Knowledge Sharing Pattern Language that covers several important knowledge sharing practices in a software development organization.

The process for creating and evaluating Knowledge Sharing Pattern Language is one of the results. It is a process of creating an organizational pattern language that is applicable also in other contexts. The resulting Knowledge Sharing Pattern Language never will be complete. A pattern language needs to mature and be improved constantly in real-life use. This makes the process of creating a pattern language important.

The validation of the KSF was made in industrial case studies. The resulting Knowledge Sharing Pattern Language was validated through justifying the single patterns separately, evaluating the coverage of the language and evaluating the applicability of the Pattern language in an evaluation workshop.

**Keywords**: software engineering, knowledge management, knowledge sharing, patterns

# Preface

I have had the opportunity to follow closely and to take part in software engineering work for many years. I have seen that similar problems repeat themselves in these environments. This idea of similar, recurring problems has been the starting point for the long journey with this study. At the end of the year 2001, I was accepted as a post-graduate student to the Tampere University of Technology. The most productive year was 2006 when I had part-time funding for doctoral studies of industrial PhD students from the Academy of Finland. After that year, most of the effort has been evaluating and refining the results as well as to writing and rewriting this thesis.

Since the journey has lasted several years, I have had the honor to meet and work with many people. I express my gratitude to all of them especially mentioning my supervisor Professor Kai Koskimies. He has been patient and always available with guidance when it was necessary. He read several versions of my writings and helped me revise them appropriately. I want to express my sincere and deep gratitude to him.

I thank Professor Tomi Männistö from the Helsinki University of Technology and Professor Markku Tukiainen from the University of Joensuu for reviewing this thesis and for their constructive feedback and comments on the manuscript.

Special thanks are due to Dr. James O. Coplien and to Professor Ilkka Haikala for their feedback on my thesis, to Professor Tarja Tiainen for her encouragement, to Emeritus Professor Pertti Järvinen for supporting this journey in many ways, to Dr. Petra Bosch-Sijtsema for the discussions we had at the beginning of the work, to Development Manager Antti Välimäki for co-authoring articles and to Elder Ralph Larson who reviewed the language of one of my latest versions of this thesis.

This thesis is a monograph, but the major results have been published in several articles. The Tampere Graduate School in Information Science and Engineering (TISE) has funded my travels to several conferences where these articles have been presented. In addition, I wish to thank my previous employer, Teleca, for allowing me to study their software engineering projects. Teleca and my current employer the Hospital District of Southern Ostrobothnia have been crucial elements in the success of this project.

Finally, all this would not have been possible without the strong support of the closest people in my personal life. This journey has not been easy for them because I have worked on this in addition to doing my regular job. My husband, Timo, especially, has had to be very flexible in many ways as well as have my children Teemu and Elina. My parents, Aimo and Tuulikki, have constantly encouraged me and been interested in how I am progressing. Also my mother-in-law was proud of me for doing this, but, sadly, she was not able to see the day this work was completed.

I have been really happy to have all you people around me. I express my sincere thanks to all of you.

Kauhajoki, October 11, 2009 Sari Vesiluoma

# Contents

# List of Figures

# List of Tables

# Part I Foundation

*This part introduces this thesis, the research questions, methods and main contributions. It also gives the background introducing the basic underlying theories regarding to knowledge sharing, knowledge management in software engineering and patterns.*

# 1 INTRODUCTION

This chapter describes the motivation for this study, explains the research questions, introduces the main contributions and maps the research questions with the contributions. In addition, the research method and approach are described. Finally, a brief overview of this thesis is given.

## 1.1 Motivation

This research has its origins in many problems identified by the author in her work over years as a process improvement specialist in software development companies and in public sector buying software development services. One part of this work has been the close study of several software engineering projects having problems. Most of the problems take place in different human interfaces. An example could be problems in defining and communicating requirements for a project together with a customer, or a project manager forgetting to inform a project team member about an approved change resulting in problems in software integration. According to Rus and Lindvall (2002) and Verner and Evanco (2003) it is very often reality in software development that a project team does not benefit from existing knowledge. Team members will repeat mistakes even though some individuals in the organization know how to avoid them. This emphasizes the importance of good knowledge sharing in software engineering work.

Software engineering is one of the most knowledge intensive professions (Handzic 2003). Many software development organizations face problems identifying the content, location, and use of their knowledge due to inadequate knowledge management (Rus and Lindvall 2002). Improved use of knowledge is the basic motivation for applying knowledge management in software engineering.

Knowledge management has been studied extensively and some research has been undertaken in the field of software engineering (e.g. IEEE 2002 and Aurum et al. 2003). Software engineering work is applying proper knowledge management to various challenges. One challenge, for example, would be retaining continuity of knowledge in a rapid staff-turnover environment, especially when the staff's software engineering knowledge is composed of a very complex combination of different layers of expertise (Edwards 2003).

Handzic (2003) defines three main reasons to start knowledge management initiatives in organizations. Those are: minimizing risk, seeking efficiency and enabling innovation. Minimizing risk approach refers to avoiding problems. That could mean, for example, not repeating mistakes but identifying potential mistake risks (e.g. based on lessons learned), and initiating risk mitigation activities to avoid those. Seeking efficiency approach refers to avoiding unnecessary duplication and reducing costs. An example of this could be the reuse of results of earlier projects, such as reusing requirements, documents, designs, components, or procedures. Efficiency could also be achieved through reducing the amount of rework e.g. through better availability of existing knowledge. Enabling innovation approach refers to knowledge creation,

which is mostly left out in this study. Minimizing risk and seeking efficiency in software engineering work are focused in this work.

Knowledge management covers different kinds of knowledge processes. These vary slightly according to the source. One subset of knowledge management processes could be defined as knowledge sharing processes. As noted by Turner and Makhija (2006), the issues associated with knowledge sharing have received little systematic attention compared to the knowledge creation issues. Rus and Lindvall (2002) remind us of the importance of knowledge sharing in utilizing the individual knowledge in the organization. They say: ”Large organizations can not rely on informal sharing of employees’ personal knowledge. Individual knowledge must be shared and leveraged at project and organization levels.” (Rus and Lindvall 2002, p. 27).

Very often, knowledge sharing is supported through establishing a knowledge base or some other database system. The real missing part is normally not the tool, but the human action. Even with sophisticated tools real knowledge sharing requires human actions. Software engineering is a group activity requiring much communication (transfer of knowledge) (Johnson 2006, pp. 13-15), collaboration (mutual sharing of knowledge) (Lindvall and Rus 2003), and coordination to ensure efficiency of the work. In this study, communication, collaboration and coordination are defined to be important actions in efficient knowledge sharing.

Why knowledge sharing does not take place (e.g. cognitive and motivational factors, Hinds and Pfeffer 2003) would be an important question. In this study the scope, however, is more to define when it should exist and seeking support for such situations. Existing literature does not give much support for that. The presented models of knowledge management or knowledge sharing are very often too general for immediate practical usage (Reifer 2002), or too tool/technology oriented (e.g. Tiwana and Ramesh 2001 or Dingsøyr and Conradi 2003). Improving knowledge sharing in companies is often initiated through building central databases, spending considerable amounts of money for technology, and then starting to wonder why no one is contributing to or retrieving knowledge from the database (Dixon 2000, p. 2). The knowledge sharing models introduced in software engineering (e.g. Oliver et al. 2003) have likewise a rather general character, failing to describe where knowledge sharing should take place. Some introductory models, like Experience Factory (Basili et al. 1994), are too specific for creating more holistic understanding about what kind of knowledge sharing should exist in software engineering.

## 1.2 Research Question

The research question in this study is: *How to understand and support knowledge sharing in software engineering?* This research question is broken down into two sub-question areas (Figure 1).

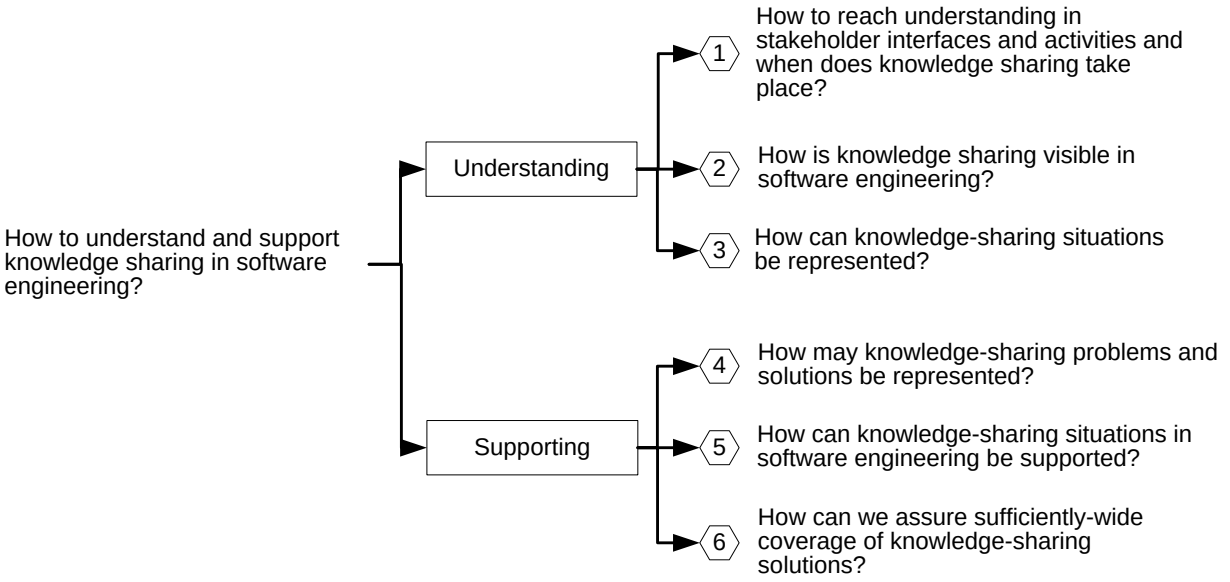



Figure 1. Research Question Tree.

To support knowledge sharing in software engineering, we first need to understand the current situation. Humphrey (e.g. 1998) uses the term "unfreezing" for actions meant to make existing problems visible. Through understanding, or unfreezing, the motivation is created for improvement activities. The need to understand what knowledge sharing means in practice arose after the importance of knowledge sharing problems among all the problems in software engineering projects became apparent in the environments of the researcher. This created the first three sub-questions. The first research question is *How to reach understanding in stakeholder interfaces and activities and when does knowledge sharing take place?* This question aims at understanding better the general environment for knowledge sharing in software engineering, and identifying existing or missing knowledge sharing in a certain environment. The second question *How is knowledge sharing visible in software engineering?* means gaining understanding about how knowledge sharing takes place in software engineering in general. The third question *How may knowledge-sharing problems and solutions be represented?* aims at describing the environment for a single knowledge sharing activity.

Eventually, the aim is to support work practices in an organization in order to have efficient knowledge sharing. Support is normally initiated based on some actual or potential problems or risks. Solutions are required to support the practice. The fourth question, about *representing knowledge sharing problems and solutions*, is related to this. The fifth question *How can knowledge-sharing situations in software engineering be supported?* targets attempts to find practical ways of introducing the solutions to support software engineering practices. The last question is *How can we assure sufficiently-wide coverage of knowledge-sharing solutions?* and targets to ensure adequate support coverage.

## 1.3 Research Methods and Approach

This is a constructive, theory-creating study following the principles of design science (Hevner et al. 2004). Also the phases defined by Eisenhardt (1989) are used when evaluating the first constructs in an industrial case study.

Van Aken (2005) defines the academic research objectives in design science as being of a more pragmatic nature than in other academic research. Research is solution-oriented and can be seen "as a quest for understanding and improving human performance" (van Aken 2005, p. 22). Design science suits this study well, because of its pragmatic nature and origin (research made based on practical problems and needs), and because the main target is to improve work results through understanding and supporting knowledge sharing in software engineering.

Hevner et al. (2004, p. 80) define a framework for information systems research (Figure 2) with which they describe an environment for design science research. This framework is applicable in this study, as well. Hevner et al. (2004) explain the framework the following way. The environment (Figure 2) defines the problem domain consisting of people, organizations and technology. Research is implemented in two phases, first, creating/building an artifact, theory etc. and then justifying/evaluating it. The knowledge base provides raw material for the research. Foundations are utilized during the development, and the methodologies are utilized in justification/evaluation. Rigor comes through appropriate application of foundations and methodologies. Relevance is assured through framing research activities to address business needs.

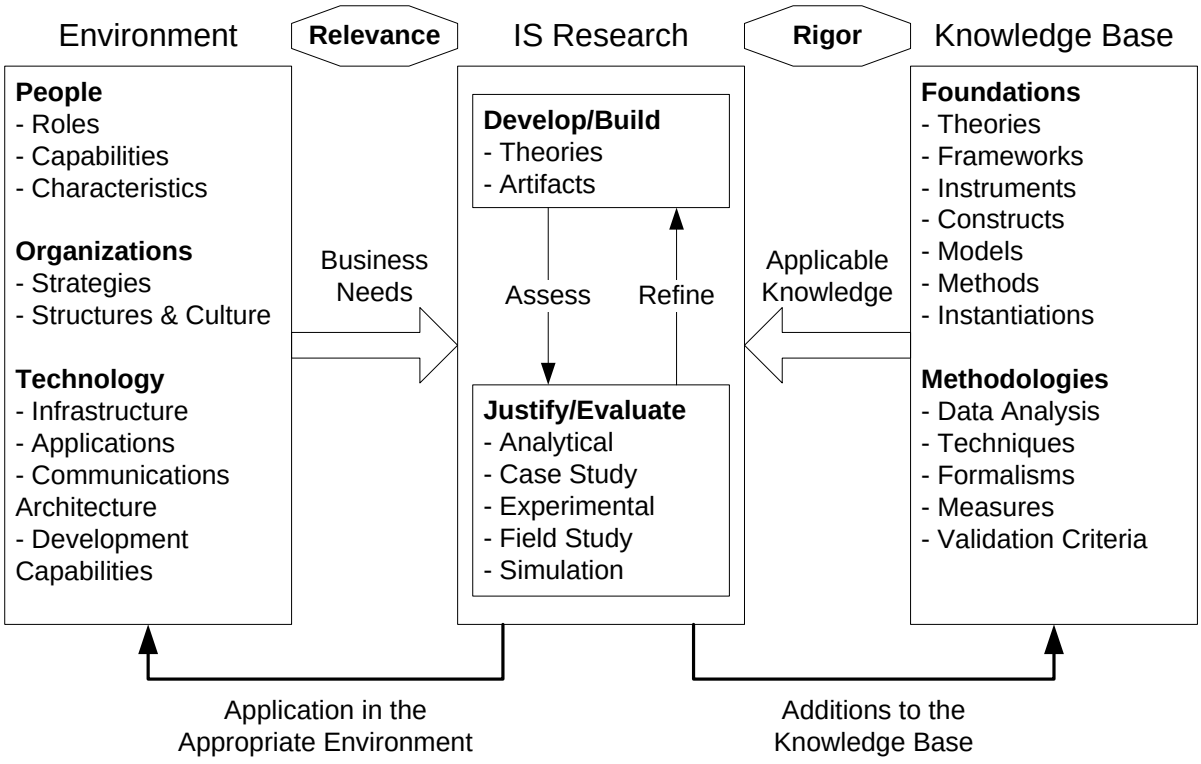



Figure 2. Information Systems Research Framework (reproduced from Hevner et al. 2004, p. 80).

Relevance of this study comes from a practical need to improve results of software engineering work. As stated before, in section 1.1, the origin of this research has been real-life problems in software engineering projects. Those problems raised a need to support knowledge sharing in software engineering. To support knowledge sharing, the knowledge-sharing environment and situation in it must first be understood. Thus, the research approach in this study consists of a path through understanding to supporting as described in Figure 3.

The research approach is introduced at more detail level in the form of a concept diagram (Figure 4). This diagram is an extension from Figure 3. Some numbers are added in order to clarify the research approach showing the constructs or alike created. Two of the constructs, number one and number five, are bolded and those are the main constructs resulting from this study. Similarly as in Figure 3, the concepts are divided into two groups: Understanding and Supporting. The same division is used in this thesis. The dashed lines in Figure 4 introduce the grouping between the understanding and supporting parts of this thesis.

Knowledge Sharing Framework (KSF) (number 1 in Figure 4) is developed for creating understanding of knowledge sharing n general and as a tool to be used when evaluating knowledge sharing situation of an organization. KSF utilizes three dimensions (2): knowledge-sharing interfaces, software-engineering activity types and project lifecycle phases. KSF is validated with project studies (case study). Use of KSF results in visual presentations, called KSF Profiles (3). Those describe the knowledge-sharing status at a studied environment based on the dimensions (2) of the KSF (1). A knowledge-sharing environment is further defined through creating the Knowledge Sharing Domain Model (4).

The advice of Handzic (2003) is followed by turning attention to new ways and tools for improving knowledge sharing in the software development process. Because different kinds of projects need different software development process models, it would be no use to introduce yet another, new, process model. In this study, instead, the target is to enrich current process models with better knowledge-sharing properties. A new technique, knowledge sharing patterns (5), is introduced to accomplish this. Pattern format is used in this study as a tool for introducing knowledge-sharing support for software engineering. Possible alternatives for it are not evaluated but its usability is shown with the Knowledge Sharing Domain Model (4).

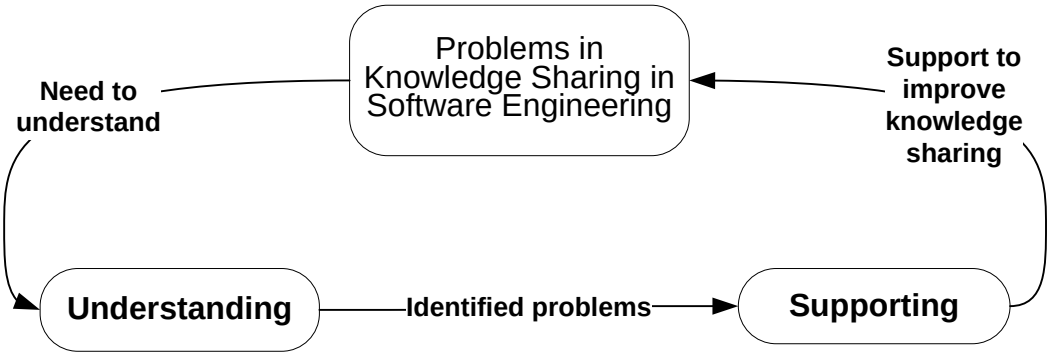



Figure 3. Research Approach through Understanding to Supporting.

Knowledge sharing pattern development process itself is one of the contributions. It is a process for creating an organizational pattern language (6). This process has utilized different sources, including the case studies. The patterns cannot be invented from scratch, but they must be based on existing practices and widely approved methods.

To assure wide enough coverage a Knowledge Sharing Goal Tree (7) is introduced defining targets for patterns. The main attributes utilized in it are the knowledge-sharing interfaces and also the target knowledge types derived from the software engineering activity types. The set of patterns introduced, the Knowledge Sharing Pattern Language (8), covers a wide range of different knowledge-sharing situations in practice. Together, all defined knowledge-sharing patterns give strong guidance to a software engineering organization for support of knowledge sharing. It must be noted that at the time of writing this thesis, the patterns are still prototype patterns in the sense that they need further improvement based on community feedback. Finally, the resulting Knowledge Sharing Pattern Language has been evaluated through a set of evaluation techniques (9), including a workshop-based validation technique.

In Figure 5 the key elements of this research approach are mapped with the IS Research Framework of Hevner et al. (2004). Abbreviation *KS* used in Figure 5 refers to *knowledge sharing*. As can be noticed, the contributions of this study are related to both the development and evaluation phases and to the knowledge base (marked as *Resulting* in the Knowledge Base column in Figure 5) of the IS Research Framework.

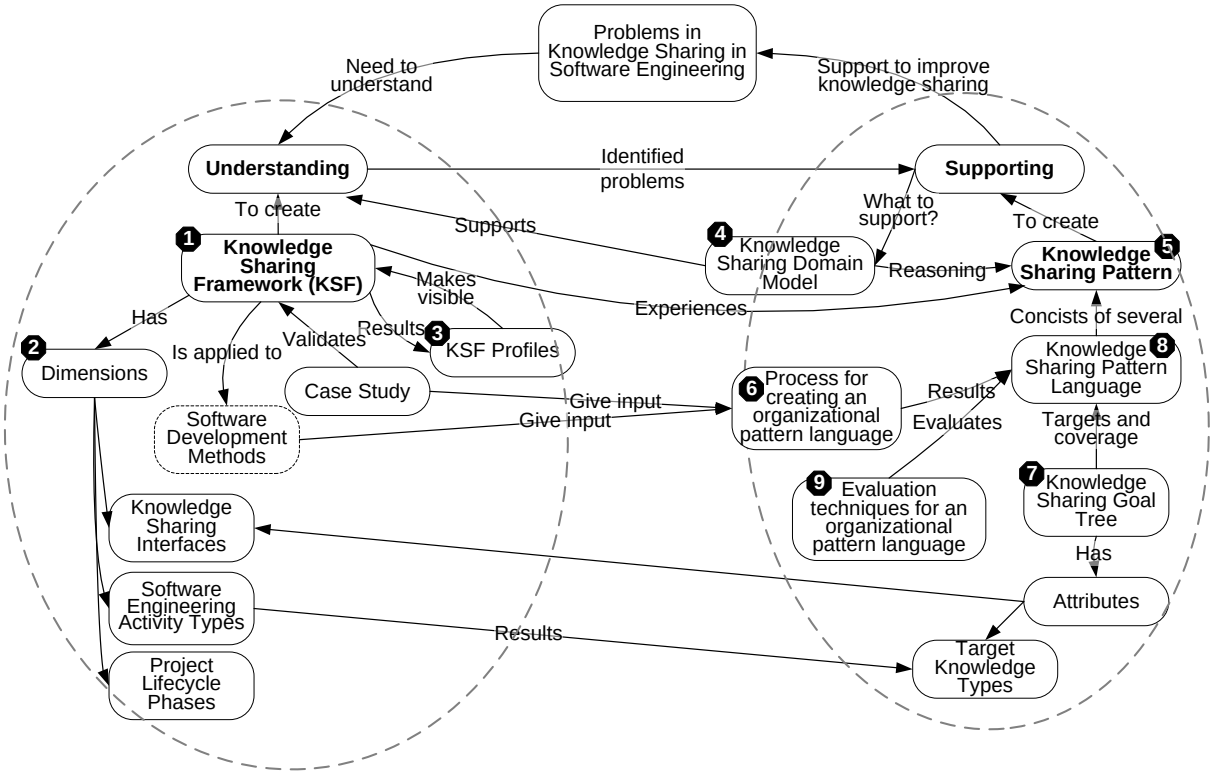



Figure 4. A Concept Diagram of Thesis Topics.

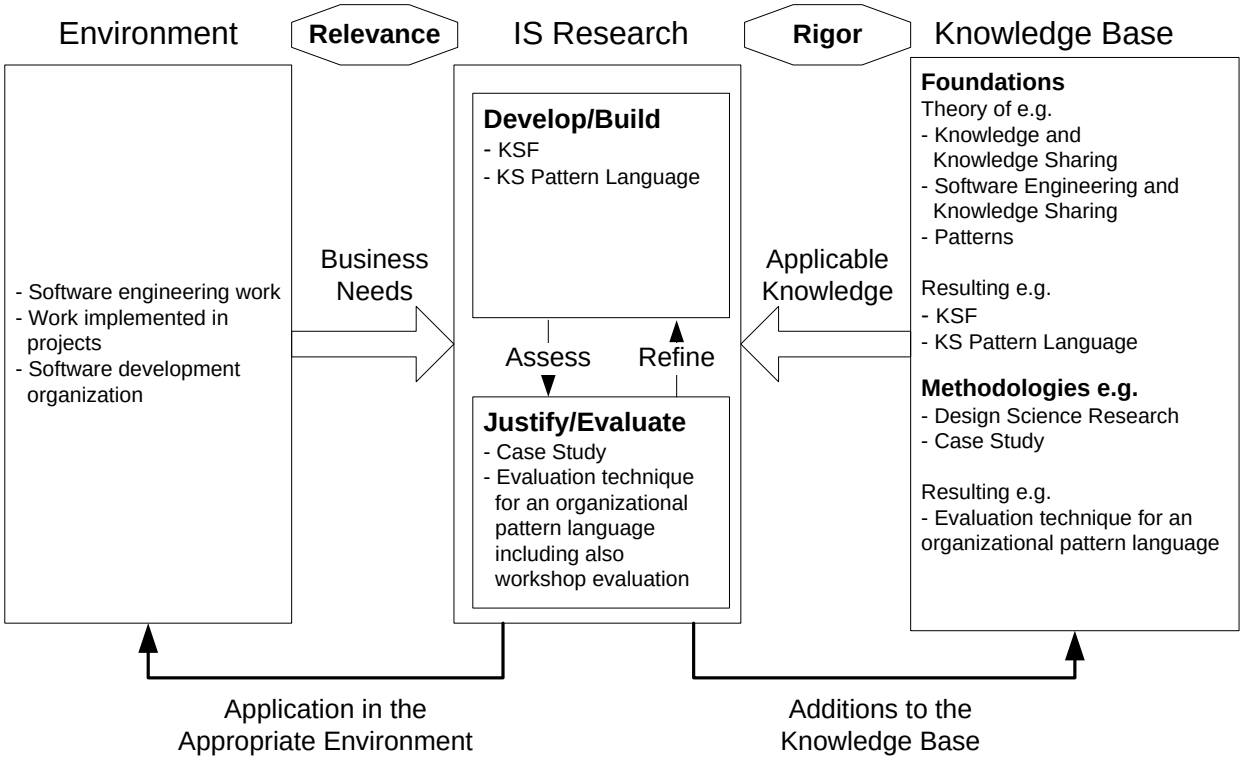



Figure 5. Mapping some key elements of this research approach and the IS Research Framework (adapted from Hevner et al. 2004, p. 80).

In this study the environment is software-engineering work, in project–oriented, software development organizations. The main results from the develop/build phase are Knowledge Sharing Framework (KSF) for understanding the knowledge-sharing situation and Knowledge Sharing Pattern Language supporting knowledge sharing in software engineering work. The justification/evaluation is introduced in Part IV (Evaluating). Chapter 2 (Background) introduces the three main foundations utilized in this research. Most of the results from this study are also contributions to the foundations. A case study is used as a methodology to justify/evaluate part of the results. Selected software development methods are used as one source in creating understanding of knowledge sharing in software engineering. In this study, when used that way, the software development methods are classified more like foundations than methodologies. Section 12.3 summarizes the use of the design science research methodology in this study.

## 1.4 Contributions

The contributions of this research are versatile. First, it introduces new concepts and solutions for the area of knowledge sharing. Second, it produces tools to understand and support knowledge sharing in software engineering work. Third, this research is an example of a process of producing a pattern language based on practical needs. The contributions of this study are summarized related to the research questions in Table 1.

Table 1. Results from the research.

| | Research Question | Result | Explanation |
|---|---|---|---|
| **Understanding** | 1 How to reach understanding in stakeholder interfaces and activities and when does knowledge sharing take place? | **Knowledge Sharing Framework (KSF)** | A framework for explaining and studying the knowledge sharing environment of a software engineering project business organization. |
| | | KSF Profiles | Visual presentation of a knowledge sharing situation in certain environment. |
| | 2 How is knowledge sharing visible in software engineering? | Comparison of software development approaches based on the KSF | Analysis of differences and similarities between selected traditional (plan driven), agile and product line based methods. |
| | 3 How can knowledge-sharing situations be represented? | Knowledge Sharing Domain Model | Definition of single knowledge sharing activity and its environment in software engineering. |
| **Supporting** | 4 How may knowledge-sharing problems and solutions be represented? | Knowledge sharing pattern concept | A certain type of pattern, a knowledge sharing pattern, is introduced. |
| | 5 How can knowledge-sharing situations in software engineering be supported? | Process for creating an organizational pattern language | A systematic process applied in the creation of the Knowledge Sharing Pattern Language. |
| | | **Knowledge Sharing Pattern Language** | An organized set of knowledge sharing patterns. |
| | 6 How can we assure sufficiently-wide coverage of knowledge-sharing solutions? | Knowledge Sharing Goal Tree | Specification of knowledge sharing objectives. Used for assuring the coverage of the patterns. |
| **Evaluation** | | Evaluation techniques for an organizational pattern language. | A systematic way of evaluating an organizational pattern language applied to evaluate the Knowledge Sharing Pattern Language. |

Most of the contributions in Table 1 are different intermediate results and artifacts needed to create the main contribution Knowledge Sharing Pattern Language. From the perspective of design research, the main artifacts resulting from this study are Knowledge Sharing Framework (KSF), and the Knowledge Sharing Pattern Language (bolded in Table 1). KSF is utilized, for example, for studying the knowledge-sharing approach of selected software development methods being one of the contributions.

In addition, several other artifacts are produced, e.g. a set of the KSF Profiles, a Knowledge Sharing Domain Model, a knowledge sharing pattern concept, and a Knowledge Sharing Goal Tree. These all contributions also benefit the knowledge-base foundations as defined in design science by Hevner et al. (2004).

The evaluation of the main results is introduced in Part IV (Evaluating). The technique of evaluating the Knowledge Sharing Pattern Language is a new way of evaluating and validating organizational patterns and pattern languages and one of the contributions of this study.

Most important results have been published separately. The Knowledge Sharing Framework (KSF) was published in the 15th European Conference on Information Systems (Vesiluoma 2007a). The Knowledge Sharing Pattern Concept was published in ENASE[1] and in the Journal of

[1] ENASE = Evaluation of Novel Approaches to Software Engineering, September 2006, Erfurt, Germany.

International Transactions on Systems Science and Applications (Vesiluoma 2006). The Knowledge Sharing Pattern Language was introduced in SQM 2007 conference[2] and in book *Software Quality in the Knowledge Society* (Vesiluoma 2007b). The scenario-based evaluation of pattern languages was introduced in PROFES 2009 conference[3] (Välimäki et al. 2009).

In addition, some intermediate results have been published. Those include, for example, first thoughts of how to develop knowledge-sharing patterns (Vesiluoma 2005) and more advanced thinking about the approaches of knowledge sharing in different software development methods (Vesiluoma 2007c).

## 1.5 Overview of the Thesis

This thesis is divided into four parts. The first part includes this introduction and the theoretical background. The second part introduces how the understanding was created. It introduces e.g. the Knowledge Sharing Framework and how it was applied in practice. Based on created understanding Part III then introduces how knowledge sharing can be supported. There the concepts of Knowledge Sharing Pattern and the Knowledge Sharing Pattern Language are presented. Part IV is the evaluation of the whole research. Finally, Part V (Closure) discusses the related work and reviews this study from different perspectives. Part V also summarizes the conclusions.

[2] Software Quality Management, SQM 2007, Tampere, Finland.

[3] Product-Focused Software Process Improvement, PROFES 2009, Oulu, Finland.

# 2 BACKGROUND

Main background concepts are introduced in this chapter. Since supporting knowledge sharing is the final target, the theory behind knowledge in general and knowledge sharing is required. Next, software engineering, the target environment for knowledge sharing is briefly studied. There, some features of software engineering organizations are highlighted, the utilization of knowledge management in software engineering is introduced, and knowledge sharing and reusing are briefly discussed.

Finally, theory related to patterns is introduced based on the literature. The background of patterns is introduced, and some basics are given from pattern formats in general. Also some guidance is given for pattern writing, especially for design patterns, but that is later in this thesis applied also for organizational/process patterns that the knowledge sharing patterns are.

## 2.1 Knowledge and Knowledge Sharing

### 2.1.1 Knowledge

Nonaka (1994) defines knowledge as justified true belief. It means that people justify the truthfulness of their observations based on their other observations of the world (Nonaka et al. 2006). Knowledge is created by individuals and the same information flow can create different knowledge and meanings for different individuals. The difference between information and knowledge is that information is a flow of messages to a person whereas knowledge is created individually based on the commitments and beliefs that a person has had (Nonaka 1994). Dixon (2000, p. 13) says that *information* is defined as *data* that is sorted, analyzed, displayed, and is communicated. He defines *knowledge* "as the meaningful links people make in their minds between information and its application in action in a specific setting."

In addition to the hierarchy of data-information-knowledge several authors suggest also category wisdom (e.g. Zelany 1987 and Ackoff 1989). In this study, like in Davenport and Prusak (1998, p. 2), higher-order concepts like wisdom, have been included in the category knowledge (in data-information-knowledge hierarchy) for practical purposes.

Rather than regarding knowledge as something people have, Blackler (1995) suggests that it is referred to as knowing regarding something people do. He also says that this kind of an approach draws attention to the need to study ways in which the systems which mediate knowledge and action are changing and might be managed. Cook and Brown (1999) define knowledge as a tool of knowing and knowing as an aspect of our interaction with the social and physical world. They see knowing as epistemic work that is done as part of action or practice. Knowing is dynamic, concrete and relational. Knowledge about how to ride a bicycle is static and exists even while not riding. Knowing is needed while riding the bike. Knowledge is partly tacit and partly explicit. According to Polanyi (1966) explicit knowledge is codified knowledge

which can be transferred in the form of systematic language, and tacit knowledge then has a personal quality and is deeply rooted in action, commitment and involvement in a specific context.

Von Krogh et al. (1994) say that everything known is known by somebody. Individuals have private knowledge that is the basis for organizational or common knowledge. When private knowledge is shared among organizational members it has potential to become organizational knowledge. Dixon (2000, p. 11) defines common knowledge as knowledge that resides in an organization, being the knowledge that employees learn from doing the organization's tasks. He also defines common knowledge as the "know how" rather than the "know what" of school learning.

Knowledge becomes organizational knowledge through time-consuming collective acceptance (Huysman and de Wit 2003). Huysman and de Wit (2003) use an example of technicians finding a new way to fix a machine. New operational knowledge remains local until it is accepted by the organization e.g. by including it into the organizational stories and newcomer trainings.

Based on literature, Orlikowski (2002) has found two perspectives on organizational knowledge. One perspective sees organizational knowledge as "a stock of resource to be created and managed" and the other sees knowledge "to be a product of dispositions and collective practice" (Orlikowski 2002, p. 269). The first one tries to find out different types of knowledge and the other one concentrates more on knowledge as knowing. Orlikowski highlights the importance of the latter and notes that knowing is constituted and reconstituted every day in practice being necessarily provisional. She sees organizational knowing as an enacted capability resulting that core competences or capabilities of the organization are constituted every day in the ongoing and situated practices of the organization's members.

Organizational knowledge is created through knowledge conversion (Nonaka et al. 2006). Nonaka defines four modes of knowledge creation. These four knowledge creation modes are at the same time knowledge conversion modes. Socialization (from tacit to tacit knowledge) can take place, for example, in on-the-job training, where an individual can acquire tacit knowledge without language. Combination (from explicit to explicit knowledge) means, for example that new knowledge is created through reconfiguring existing information, combining different bodies of explicit knowledge. Metaphors play important roles in externalization (from tacit to explicit knowledge) and internalization (from explicit to tacit knowledge) is close to what learning means.

Cook and Brown (1999) do not agree with Nonaka (1994) that knowledge could be converted from one type to another, especially that knowledge could be converted between tacit and explicit knowledge. They pose that knowledge should not be treated as being essentially of one kind. Instead they introduce four forms of knowledge (Figure 6), which consist of the combinations of two categories: individual/group and explicit/tacit. The individual knowledge, as Cook and Brown (1999) has defined, refers in this thesis to the private knowledge of an individual and the group knowledge refers to the organizational knowledge.

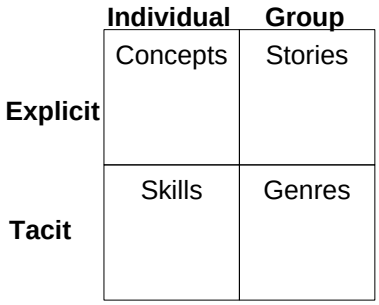



Figure 6. Four Forms of Knowledge (Cook and Brown, 1999).

Cook and Brown (1999) define these four forms of knowledge as following: Concepts are individual things an individual can know, learn and express explicitly (examples: concepts, rules, equations that typically are presented explicitly and used by individuals). Stories are explicitly expressed things that are typically used, expressed or transferred in a group (examples: stories about how work is done or about famous successes or failures, as well as the use of metaphors or phrases that have useful meaning within a specific group.). Skills are examples of tacit knowledge possessed by individuals (examples: a skill in making use of concepts, rules and equations or a "feel" for the proper use of a tool or for keeping upright on a bike.). Genres then refer to tacit knowledge possessed by groups, the group's "sense" of what the mission statement means (it can mean different things in different groups). It can be used only in the context of group practice.

In addition to tacit/explicit and the four forms of knowledge by Cook and Brown (1999), knowledge management literature includes several different kinds of typifications of knowledge. Some of those are included here. First is Sveiby and Lloyd's (1987) two main types of knowledge (their original term: knowhow) in knowledge intensive companies. Those are professional knowledge and managerial knowledge. With professional knowledge they refer to the core knowledge of the knowledge intensive company. The success of a company depends ultimately on the abilities of its professionals. In addition to professional knowledge, managerial knowledge is needed. They define managerial knowledge to be the "ability to preserve and enhance the company's value".

Based on a review of organizational learning literature, Blackler (1995) has defined five types of knowledge: embrained, embodied, encultured, embedded and encoded. Embrained knowledge is abstract knowledge dependent on conceptual skills and cognitive abilities. Embodied knowledge is action oriented and likely to be only partly explicit. Encultured knowledge refers to the process of achieving shared understandings. Embedded knowledge resides in systemic routines and encoded knowledge is information conveyed by signals and symbols. When linking this to the personal/organizational knowledge as discussed in this thesis, embrained and embodied knowledge seem to be very personal types of knowledge, while the others could be thought to be more like organizational knowledge.

Brown and Duguid (2001) have found "sticky" and "leaky" types of definitions of knowledge in literature. Sticky discussions focus on the challenge of moving knowledge inside organizations. Leaky discussions concentrate on the external and undesirable flow of knowledge, especially the loss of knowledge across the boundaries of a firm to competitors.

To summarize, in this study knowledge is defined as something people create individually (Nonaka 1994), is useful for them in certain contexts (i.e. situation dependent, Blackler 1995) and has both tacit and explicit dimension (Polanyi 1966). Also it is recognized that knowledge possessed by a person, group or organization is very much situationally dependent, and it develops continuously (Blackler 1995) based on new knowledge gained. In this study the target is to utilize better existing personal or organizational knowledge through sharing it in an organization to recreate the organizational knowledge.

The terms common knowledge and organizational knowledge are near to each other by definition. In this study organizational knowledge refers to common knowledge residing in an organization. The term organization does not necessarily mean one complete company. Depending on the situation it can also mean one unit or team of a company.

### 2.1.2 Defining Knowledge Sharing

Von Krogh et al. (1994) define two conditions that need to be satisfied to have knowledge connections in an organization: availability of relationships and self-description. The relationships in the organization enable knowledge connections. Examples of formal relationships are connections through organizational structures and reporting relationships. The self-description provides criteria what to count as knowledge, and what to pass through the connection. It also defines some part of the communication as noise that should not be connected. The self-description prevents the organization from drowning in knowledge complexity.

Existing relationships and right self-description are required to ensure knowledge sharing, but how then will the knowledge sharing actually takes place? To explain it, the theory of Boland and Tenkasi (1995) is used here. They explain that a knowledge-intensive company relies on multiple specialties and knowledge disciplines to achieve their objectives. A community of knowing is defined to be a community of specialized knowledge workers. A knowledge intensive company consists of several of these overlapping communities of knowing.

In this study, a community of knowing consists of from one to several persons having a similar knowledge background. One person could belong to several overlapping communities of knowing. For example, a software architect could belong to a software architecture community of knowing, but at the same time possess some special range of competences in a certain technology branch.

The basis for knowledge sharing within and between communities of knowing is the process of perspective making and perspective taking (Figure 7). With perspective making, Boland and Tenkasi (1995) mean the process where a community of knowing develops and strengthens its own knowledge domain and practices. With perspective taking, they mean the communication required for taking the knowledge of other communities into account. In this study, knowledge sharing is defined to be this process of perspective making and perspective taking.

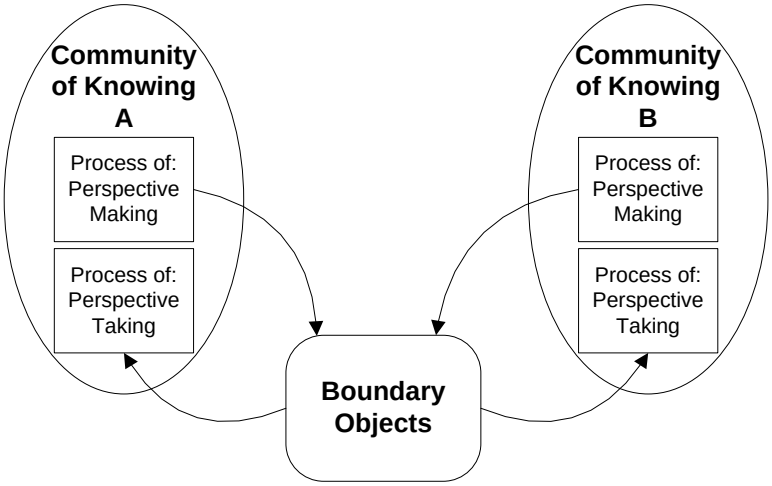



Figure 7. Process of Perspective Making and Taking according to Boland and Tenkasi 1995.

To implement perspective taking, one's understanding must be made somehow visible. This visible understanding is named as a boundary object (Figure 7). Boland and Tenkasi (1995) give examples of boundary objects, such as physical models, spreadsheets and diagrams. Star and Griesemer (1989) have defined a boundary object to be an object inhabiting several intersecting social worlds, something that maintains a common identity across several parties. Boundary objects are tools for cooperation and are a way to make viewpoints of some parties visible. When viewpoints are made visible they can be negotiated, debated and triangulated (Star and Griesemer 1986). In general, a boundary object can also be a concept or term for which a shared meaning is created that is used in transferring knowledge between communities of knowing.

Dixon (2000, p. 11) introduces an example of knowledge sharing through cycles of translating experience into common knowledge and then leveraging that common knowledge. The steps of the creation of common knowledge defined by Dixon are described in Figure 8 as the inner circle. The inner circle includes four steps starting with a team performing a task that result in an outcome. What follows then is the most important step of translation. The team explores the relationship between action and outcome. According to Dixon the result is common knowledge.

The second activity, also including four steps as shown in Figure 8, is according to Dixon (2000, pp. 19-21) the leveraging of common knowledge. There, a knowledge transfer system is selected, and knowledge is translated into a form usable by others. The receiving team adapts knowledge for its own use and utilizes it for performing a task, which initiates new circles. The adaptation by the receiving team can also result in innovations or stepwise improvements on what the earlier team was able to do.

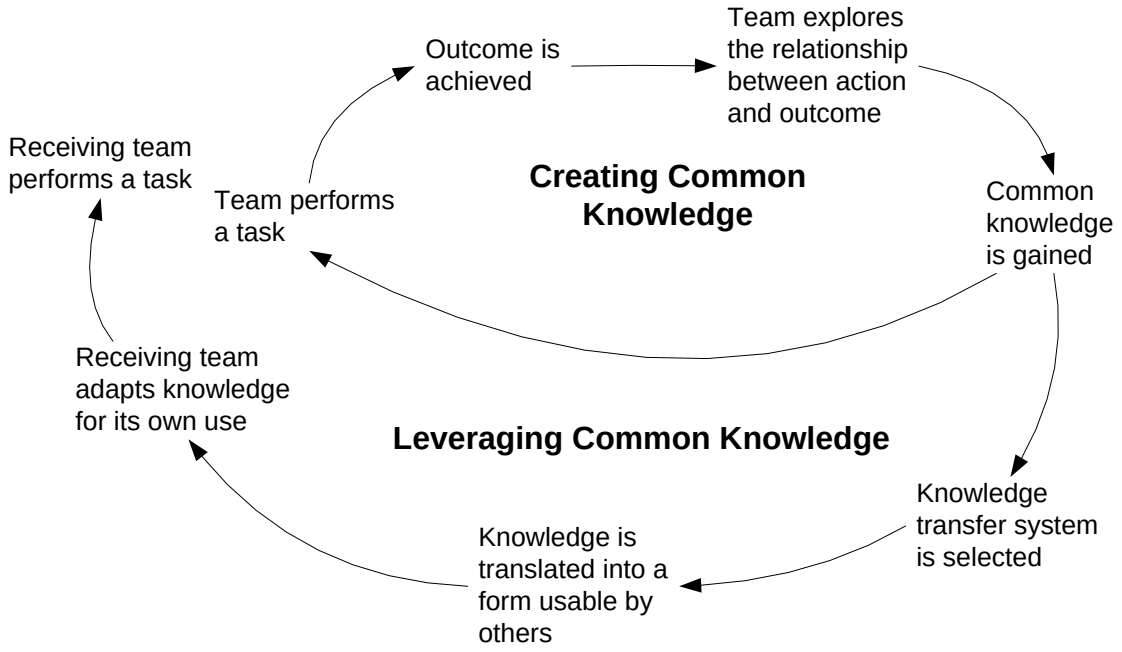



Figure 8. From Experience to Common Knowledge (reproduced from Dixon 2000, p. 20).

Utilizing the terminology of Boland and Tenkasi (1995), creating common knowledge and the first two steps of the leveraging common knowledge (selecting knowledge transfer system and translating knowledge into form required) are the perspective making process leaving common knowledge as the result. The rest of the phases of leveraging common knowledge are then an example of perspective taking process. The resulting common knowledge could e.g. be a lesson learned (documented or undocumented) that is first understood at a project team level (perspective making). The perspective taking then could be initiated e.g. in sharing the lesson learned in an organization's project manager meeting, or through utilizing documented lessons learned by some other project team in their work.

To summarize, in this study, at abstract level, knowledge sharing refers to the process of perspective making and taking (Boland and Tenkasi 1995). At operational level knowledge sharing can take place at many ways. Figure 9 introduces the abstract and operational levels of knowledge sharing and gives some examples of how knowledge sharing may take place in practice.

Knowledge sharing is defined here always to take place between two or more individuals. The individuals, however, do not need to initiate knowledge sharing and receive knowledge at the same place and time. If knowledge sharing takes place at the same time and place examples of means for knowledge sharing could be direct oral communication or observations. If the time and place are not the same for these individuals, knowledge can still be shared, but knowledge need to be made explicit (e.g. documented) as information and this information to be made available to another individual. In this case, information could be thought to be a boundary object utilized in perspective making and perspective taking process. The receiver of information processes the information and creates individual knowledge based on it (perspective making). It must be noted, that whatever the operational way of sharing is, the individual sharing knowledge cannot fully ensure that the knowledge is received exactly as wanted.

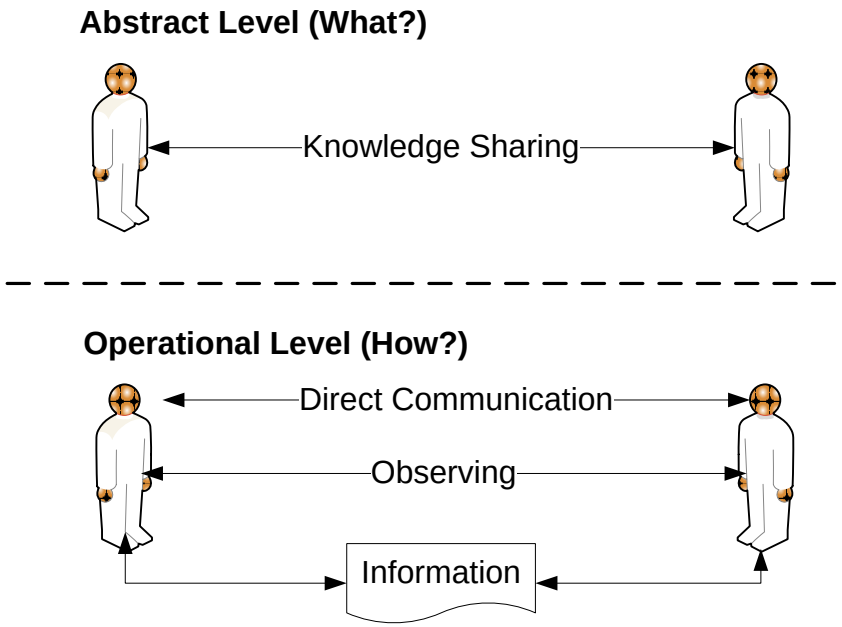



Figure 9. Knowledge Sharing Seen from Abstract and Operational Level.

### 2.1.3 Knowledge Sharing as Part of Knowledge Management

Knowledge management literature includes several descriptions of the kinds of knowledge processes that are required in organizations. From the many existing definitions, the definitions of Davenport and Prusak (1998) and Turner and Makhija (2006) are selected as examples. Davenport and Prusak (1998 pp. 51-101) define three knowledge processes: generation, codification and transfer of knowledge. They state that any organization wanting to excel at managing knowledge will have to perform these three processes well. Knowledge generation refers to activities and initiatives organizations undertake to increase their stock of corporate knowledge. Knowledge codification means putting organizational knowledge into a form that makes it accessible to those needing it. Finally, knowledge transfer refers to the transfer of knowledge from one person or group to another. This takes place whether or not this process is managed at all in an organization. According to Davenport and Prusak knowledge transfer involves transmission and absorption. Transmission means sending or presenting knowledge to a potential recipient. Knowledge must be absorbed by the receiver. Otherwise the knowledge has not been transferred.

Turner and Makhija (2006) define four critical management stages of an organization's knowledge: knowledge creation and acquisition, the transfer of knowledge to other individuals or organizational units, the interpretation of this knowledge in a manner conducive to the objectives of the organization, and application of the knowledge toward organizational goals. Knowledge creation and acquisition consists of the processes by which new knowledge is obtained by the organization. The knowledge may be newly created or acquired from external sources. After knowledge is created or acquired, it has to be transferred or disseminated to other parts of the

organization needing it or who might benefit from it. The value of the knowledge is derived from the interpretations given to it by the recipients. The a priori base of knowledge of the receiver affects the interpretation of the new knowledge. The target of all of this is to apply knowledge toward organizational ends.

In Table 2 the knowledge processes of Davenport and Prusak (1998) and Turner and Makhija (2006) are mapped to each other. The gray fields indicate areas of no direct mapping. Those include Davenport and Prusak's knowledge codification process, and Turner and Makhija's application of knowledge process.

Davenport and Prusak's knowledge generation process can be mapped to Turner and Makhija's knowledge creation and acquisition process. Those both refer to similar knowledge processes. Davenport and Prusak's knowledge transfer process maps with two Turner and Makhija's processes: knowledge transfer process and interpretation of knowledge process. The latter maps especially with the absorption part of the Davenport and Prusak's knowledge transfer.

Knowledge sharing includes a sub-group of knowledge management processes. Included processes are marked in Table 2 at the last column. The perspective making part includes knowledge codification introduced as one of Davenport and Prusak's processes. The perspective taking part includes absorption (part of the knowledge transfer process of Davenport and Prusak's) or the interpretation of knowledge when using the terminology of Turner and Makhija. To make knowledge sharing happen also knowledge transfer in between these is required. To summarize, knowledge sharing includes preparing knowledge to be shared, transferring knowledge to the receiver and interpretation of knowledge by the receiver.

Table 2. Mapping Knowledge Processes.

| **Davenport and Prusak, 1998** | **Turner and Makhija, 2006** | |
|---|---|---|
| **Knowledge generation** | Knowledge creation and acquisition | |
| **Knowledge codification** | | Knowledge Sharing |
| **Knowledge transfer (transmission + absorption)** | Knowledge transfer | |
| | Interpretation of knowledge | |
| | Application of knowledge | |

Knowledge transfer and knowledge sharing are terms that are not very well defined, or better said, these are defined many ways. For example, Arogate and Ingram (2000, p. 151) define knowledge transfer as "the process through which one unit (e.g. group, department, or division) is affected by the experience of another." This definition is wider than knowledge transfer used in the definitions of Davenport and Prusak (1998) and Turner and Makhija (2006). Van Baalen et al. (2005) have selected the term knowledge sharing instead of knowledge transfer because it succinctly refers to the social processes that are involved. This has also been the basis for the difference of knowledge transfer and knowledge sharing in this study. Knowledge sharing covers also other knowledge sub-processes than knowledge transfer (see Table 2).

## 2.2 Software Engineering and Knowledge Management

### 2.2.1 A Software Engineering Organization

Based on knowledge-work literature, Blackler (1995) introduces four categories of organizations. He uses two criteria. The first one is, whether the emphasis in the work is on contributions of key individuals or on collective endeavor, and second, whether the focus is on familiar or novel problems. These four categories are:

(i) Expert-Dependent Organizations (contributions of key individuals and focus on familiar problems)
(ii) Knowledge-Routinized Organizations (collective endeavor and focus on familiar problems)
(iii) Symbolic-Analyst-Dependent Organizations (contributions of key individuals and focus on novel problems)
(iv) Communication-Intensive Organizations (collective endeavor and focus on novel problems)

Blackler (1995) states that in software engineering the focus is on novel problems, quite often on problems that are totally new based on new technology and new application areas. So, software engineering organizations should be either (iii) symbolic-analyst-dependent or (iv) communication intensive organizations. Software engineering organizations seem to be very much based on the key individuals (iii symbolic-analyst-dependent organization), but also at the same time more and more team based (iv).

Software engineering projects cannot be implemented by one or very few persons. Complexity of software engineering projects and ever increasing time-to-market pressures are among the main reasons for that. The result is that a software engineering organization should be communication-intensive organization (iv) instead of the symbolic-analyst-dependent organization (iii). A transfer from isolated problem solving attempts (iii) into a shared activity (iv) is crucial to the effective operation of a knowledge intensive organization (Blackler 1995).

### 2.2.2 Utilizing Knowledge Management in Software Engineering

Rus and Lindvall (2002) have identified motivations for utilizing knowledge management in software engineering. They have defined two categories: motivations based on business needs and based on knowledge needs.

According to Rus and Lindvall (2002), the main motivators, from a business need perspective, are: *decreasing time and cost and increasing quality*, and *making better decisions*. Avoiding mistakes reduces the need to rework therefore decreasing required time and cost, and increasing quality. To achieve this, the process knowledge gained in previous projects needs to be applied to future projects. Rus and Lindvall comment that to make better decisions an organization needs to define processes for sharing knowledge.

When comparing these business need-based motivations to the main reasons for applying knowledge management by Handzic (2003), we find that the business need for making better decisions is very similar to the minimizing risk approach of Handzic. In like manner, the business need for decreasing time and cost and increasing quality is very similar to the seeking efficiency approach of Handzic. These two business needs could be defined as the main drivers for applying knowledge management in software engineering. Knowledge need-based motivations aim to support achieving these business needs.

From the knowledge need perspective Rus and Lindvall (2002) identify five types of motivators. They define the first one being *new technology knowledge acquisition*. Software engineering projects involve new technologies that project teams are not familiar with. The second motivator is *accessing domain knowledge*, knowledge about the domain for which software is being developed. The third motivator is the need of *sharing knowledge about local policies and practices*. New engineers need knowledge about the software development practices at the organization, existing software base and culture. This knowledge is often shared informally, being a very important aspect of the knowledge sharing culture. Formal knowledge capturing of this kind ensures that all employees can access this knowledge.

Rus and Lindvall (2002) define the fourth knowledge need-based motivator to be the need for *capturing knowledge and knowing who knows what*. The key to a project's success in software engineering is the knowledgeable employees. This means that knowing what employees know is necessary for creating a strategy to prevent valuable knowledge from disappearing and also to know who has what knowledge in order to utilize this knowledge in the best possible way. The last motivator defined by Rus and Lindvall is the need for *collaborating and sharing knowledge*. A group activity, like software engineering, would be very hard without proper communication, collaboration and coordination.

### 2.2.3 Knowledge Sharing and Reuse

According to Aurum (2003) the overall quality of software can be improved through making knowledge available and using it proficiently i.e. through sharing knowledge with relevant people. An organization and individuals could gain much more if they could share the knowledge that had been acquired in earlier projects (Rus and Lindvall 2002). Through sharing, the value of knowledge to developers is increased (Aurum, 2003).

Reuse of knowledge can be practiced at many levels. According to Sutcliffe (2002, p. 13) temporal reuse, reuse over time, is closer to evolution taking place in domains where products are incrementally improved. Reusing across domains then has larger payoffs but is more ambitious.

The objective of reuse is normally to save problem-solving effort, minimize redundant work and enhance the reliability of the work (Jacobson et al. 1997, p. 6). Potential targets for reuse vary from reuse of single artifacts or knowledge to product line based systematic approaches. Barnes and Bollinger (1991) have given examples of potentially reusable artifacts like requirements specifications, designs, code modules, documentation, test data and customized tools. FAST method (will be introduced in Section 4.2.3) is one example of a product line based systematic reuse approach. Reuse of artifacts or product line based reuse could be defined as reuse of technical artifacts.

A well tested software component preserves accumulated knowledge. Most probably several hours have been used to define requirements for that component in order to design, implement and finally test it. If it is possible to reuse this component later, all the preparatory work will have already been accomplished and the team applying the component later will not need to worry about problems the earlier team has already solved. Often, the competence requirements for the team applying an existing component are lower than for the original team, in other words, the knowledge of the original team can be applied throughout the application of the component.

Mili et al. (1995) have defined two general categories of reuse approaches, the building blocks approach and the generative or reusable processor approach. The first one refers to the reuse of software artifacts. The latter, the reusable processor approach, is based on reusing processes. Standard operating procedures in an organization are one example of process reuse. Standard operating procedures capture many best practices that need to be shared in an organization. They make the work more transparent therefore making it easier to collaborate knowing what to expect from the others. In addition, standard operating procedures include a common vocabulary and known boundary objects, making it possible to have shared meanings in an organization.

Sutcliffe (2002) has studied and analyzed reusing from many perspectives. He outlines the reuse process (pp. 25-26) to include: knowledge acquisition, transformation of knowledge to a form retrievable and understandable (e.g. generalizing, documenting) by the reuser, placing it to a reuse library, matching the requirements for the new application, and understanding and applying reusable component to new environment. When comparing these to the knowledge processes listed in Table 2 we can note that reusing as defined by Sutcliffe (2002) and knowledge sharing as defined here in this thesis is nearly the same process.

According to Sutcliffe (2002, p, 26) the reused knowledge can be whatever, for example, a thought, a concept, or a software component. Reuse fails very often according to him (p. 27), for example, because of missing motivation or not having well organized component libraries to find relevant components.

To summarize, knowledge reuse is one approach to knowledge sharing. In knowledge reuse the aim is to ensure that created knowledge can be shared and utilized again and again. Knowledge sharing could be thought to take place continuously everywhere, but knowledge

reuse could be defined in most cases needing some systematic preplanning or activities to have the knowledge available in a form usable to the reuser of that knowledge.

## 2.3 Patterns

### 2.3.1 Introduction

Term pattern, as it is used in this study, is introduced by Alexander et al. (e.g., 1977) in the context of building architecture. After that, patterns have been adopted as a central concept to express collections of problem-solution pairs in specific areas of software engineering.

A pattern is a context-dependent construct which includes a problem and a reproducible solution to such a problem (Alexander et al. 1977, p. x). Patterns are more useful when they are a part of a systematically organized collection. That means a pattern language which is applied in a predetermined manner. Coplien (1996, p. 23) explains the difference between a pattern and a pattern language so that a pattern language is a collection of patterns generating a system while a single pattern just solves an isolated problem.

In software development the first introduced patterns have been design patterns (e.g. Gamma et al. 1995). In addition to those, patterns have emerged also for analysis, maintenance, testing, documentation, and organizational structure (Vlissides 1998, pp. 8). Design patterns introduce different design principles aimed at structuring the resulting code. Organizational patterns involve human interactions which build and repair the organization (Coplien and Harrison 2005, p. 3). According to Coplien (1998, p. 245) "The patterns of activity within an organization (and hence within its project) are called a process. Organization, projects and process can be viewed as partial facets of each other."

In this study, the focus is on organizational patterns. Compared to design patterns, the human component is more apparent in organizational patterns. In addition to being organizational patterns, the resulting knowledge sharing patterns can also be referred to with the term process patterns. As Coplien (1998) defines, processes can be thought to be partial facets of e.g. an organization. In addition, most of the resulting knowledge sharing patterns have the process perspective aiming to improve the organization through improved software development processes.

### 2.3.2 Pattern Formats

A pattern is a piece of literature (Coplien 1996, p. 2). A set of patterns, a pattern language, normally follows a certain type of format for describing single patterns. The Alexandrian format, the format of Christopher Alexander (e.g. Alexander et al. 1977), is one of the most literary formats (Coplien 1996, p. 12). It does not utilize many sub-topics. In contrast, the Gang of Four format (Gamma et al. 1995) uses many sub-topics. Also, several other formats exist. One of those is the Coplien format. It is somewhere between the Alexandrian format and Gang of Four

format. Regardless of the format used, however, the main parts are normally present: name, problem, context and solution.

The Coplien format (Coplien 1996) is used as the basis when defining the knowledge sharing pattern format. Its parts are the following:

1. The pattern name
2. The problem
3. The context
4. The forces
5. The solution
6. The rationale
7. The resulting context

The name of a pattern is very important, because it is expected to become a part of the professional vocabulary. Coplien (1996, p. 14) recommends using a noun or a short verb phrase as the name. He defines that the problem is often stated as a question or as a challenge. The context describes the circumstances in which the problem might arise, and to which the solution applies. The forces describe the issues that pull the solution in different directions. Coplien and Harrison (2005, p. 5) define that forces together cause the problem, and the solution must balance the forces.

According to Coplien (1996) the solution explains how to solve the problem. The rationale describes why the pattern works for its purpose and the resulting context tells which forces the pattern resolves and which remain unresolved. The rationale might also point to more patterns that deserve consideration.

### 2.3.3 Pattern Writing

Vlissides (1998, pp. 145) indicates that patterns are not written in vacuums, implying that a pattern writer needs to think about several patterns, of which several are not yet written. One pattern solves one problem, but many patterns are required to solve many problems in the same domain. This affects pattern writing. It is not enough to understand one single pattern and its environment but a pattern must be in line with the whole pattern language.

Like Covey (1989) defined the seven habits of effective people, Vlissides (1998, pp. 145-152) have introduced seven habits of effective design pattern writers. Those are mostly applicable to organizational patterns. The seven habits are:

1. Taking time to reflect
2. Adhering to structure
3. Being concrete early and often
4. Keeping patterns distinct and complementary
5. Presenting effectively
6. Iterating tirelessly
7. Collecting and incorporating feedback.

Vlissides (1998, p. 146) recommends to take time for reflection every now and then. If the experiences from e.g. programming work and problems in it are collected and recorded incrementally, those can be utilized when writing patterns.

A certain structure needs to be used for every pattern (adhering to structure). In this thesis the structure is referred to with the term *pattern format*. There are several formats existing, like described in Section 2.3.2. Like Vlissides (1998, pp. 147-148) states, different environments need different pattern formats. The format can be changed during the work of describing patterns, but then the format of all existing patterns of that pattern language needs also to be changed.

Vlissides (1998, p. 148) reminds that being concrete early and often means that even though the introduced patterns need to be applicable in different environments, real life examples also are needed to understand the abstractions. Another point is that the readers must be warned about potential pitfalls as well as being introduced to the positive effects.

As stated earlier, there will not be just one pattern, but several of those, a whole pattern language. This means that the habit of keeping patterns distinct and complementary is very important. Vlissides (1998, p. 149) recommends that immediately upon starting to write patterns a separate documentation must be kept for pattern comparison and contrast evaluation.

The patterns need to be written in such a way that they are of good quality and usable in practice. According to Vlissides (1998, p. 150) presenting patterns effectively means two things: typesetting and writing style. The pattern format, or actually, the template used for describing patterns, can include the typesetting and in that way establish a certain method of pattern description. The writing style is more difficult and heavily dependent on the writer. It needs to be such, however, that the pattern is easy to understand and it becomes possible to apply it practically according to the description. The user and the requirements of practice need to be thoroughly considered when writing patterns. The writing style used has to be evaluated as well while the patterns are evaluated.

In addition to the need of time for reflection, iterations are also needed. Like Vlissides (1998, p. 151) says, the patterns are not easy to get "right". Firstly, describing a pattern following the practice where it was found is difficult, while some kind of an abstraction needs to be done. When writing one new pattern it can result changes to some already existing patterns. One could say that there will never be "perfect" patterns only iterations and improvements that could go on forever. At some point, however, the patterns need to be shown to others and really put to use in order to get feedback.

A pattern is not an innovation but it is, rather, an existing solution that is described as a pattern and subsequently shared with people needing this kind of a solution. Therefore, in practice, a pattern is a piece of restructured, existing knowledge and becomes a shared, 'way-of-working' heritage. It is not reasonable to evaluate the goodness of a pattern because of the nature of patterns or, more generally, best practices. Instead, we can evaluate whether something could be applicable or not. A pattern will or will not be found useful in practice and the pattern author needs to be ready to listen to the feedback (habit 7 of pattern writing) and to improve the pattern or to let others attempt to improve the patterns and share those further. Ideally, patterns are elaborated within the community before they are published, for example, in the context of Pattern Languages of Programs (PLoP) conferences.

# Part II Understanding

*To have understanding of the knowledge sharing situation in a certain environment, some kind of a tool is required. The Knowledge Sharing Framework (KSF) was developed for this purpose. This part introduces KSF and gives examples how it can be used in practice. Creating understanding about knowledge sharing is important when preparing for creating support for knowledge sharing. Part III Supporting continues from the results of this Part II Understanding.*

# 3 FRAMEWORK FOR KNOWLEDGE SHARING IN SOFTWARE ENGINEERING

The Knowledge Sharing Framework (KSF) has three dimensions that are introduced in this chapter and summarized as a conceptual framework. The dimensions are: knowledge sharing interfaces, software engineering activity types and project lifecycle phases.

## 3.1 Knowledge Sharing Interface

Knowledge sharing, the process of perspective making and taking, is studied in this thesis in a software engineering environment. Most software engineering work is organized as projects and is implemented in project companies. A project company consists of company structures and project structures (Artto 1998). A project company has, according to Gareis (1996), a lean base organization and a variable portfolio of projects. In Figure 10, the structure of a project company is described. It is based on the general dichotomy: base organization and projects. The base organization consists of line management, resource pool and support functions. Based on the situation, a variable number of projects exist in different phases of their life cycle. Every project has its own project team, which consists of persons from the company's resource pool, which could include suppliers and/or customers.

Software engineering work requires many stakeholders. A stakeholder is a group or individual affected by the achievement of organizational goals, and whose own needs must be noticed to satisfy the stakeholder and not to cause difficulties for the organization (Freeman 1984). Knowledge sharing takes place between stakeholders, or according to Boland and Tenkasi (1995), between communities of knowing. Here it is defined that a knowledge sharing interface exists between stakeholders when knowledge sharing takes place or should take place between these stakeholders. The arrows in Figure 10 represent the main knowledge sharing interfaces between different stakeholders in a project-oriented company.

The stakeholders in this context are individuals, project teams, the base organization, customers and other stakeholders. Based on these stakeholders, different knowledge sharing interfaces in software engineering can be defined (see Figure 10): 1) between the individuals in a project team, 2) between the project team and the rest of the organization including other projects and base organization (organization level, grey area in Figure 10), and 3) between companies (a company and its customers, suppliers and other related parties). The interface between companies is applied in this context to customer-supplier communication.

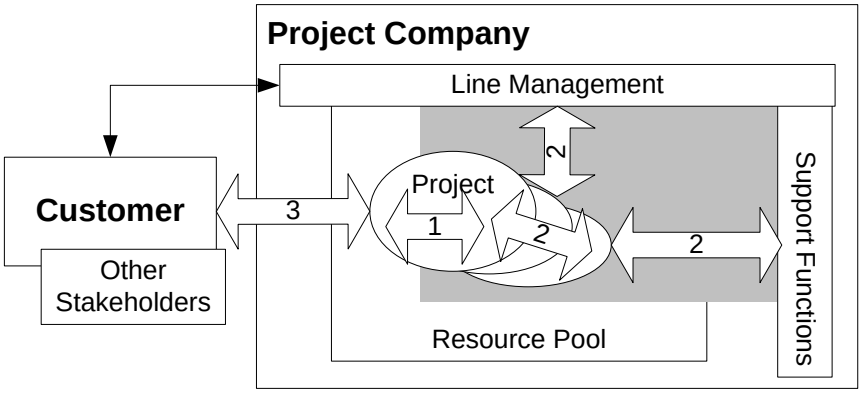



Figure 10: Structure of a Project Company

In addition to these more formal knowledge sharing interfaces, there is a lot of unofficial knowledge sharing between individuals. This occurs in all situations where an individual is sharing knowledge or communicating with another individual, even at a friendship level. This can happen at any interface. These relationships are hard to control or to take as part of the knowledge sharing structure effectively, but, they can be supported by creating opportunities for individuals to establish relationships. This is one condition of having knowledge connections as defined by von Krogh et al. (1994). Attempts can also be made to restrict this knowledge sharing, for example, by introducing information security restrictions using confidentiality agreements. Doing this would be an example of making an attempt to hinder knowledge leaking in the sense that was introduced by e.g. Brown and Duguid (2001).

Knowledge sharing has been studied in this thesis mostly from the perspective of co-located project teams. A distributed project team would bring other perspectives such as temporal and geographical distance. In one of the studied projects there were also some experts not on the main site, but that does not yet bring a proper view of a distributed software engineering setting.

## 3.2 Software Engineering Activity Types

A knowledge sharing interface is defined based on the structure and stakeholders of a project company. However, a more software engineering specific approach is still required. One possibility could be to divide software engineering work according to the traditional waterfall model (Royce 1970) phases. That division, however, is the basis for the traditional plan-driven software development processes, so some other division is required in this study, in order not to limit thinking with the vocabulary of traditional processes. As a result of this kind of thinking, in this study, software engineering work is divided into three activity types: value adding software engineering, verifying software engineering and managing software engineering.

Value adding and verifying software engineering activities have been introduced by Kylmäkoski (2006). Value adding software engineering is that part of a project where new results are gained and progress is achieved. It is the part of a project where the value is added. It is based on the idea of value added analysis activity in Business Process Improvement (e.g.

Harrington 1991). For each activity Harrington (1991) asks whether or not the activity is necessary to produce output. This separates e.g. analysis, design and implementation from quality assurance related activities like reviewing and testing. Value adding software engineering, as it is defined in this study, must be distinguished from Boehm's Value Based Software Engineering (Boehm 2006) despite the apparent similarity in their names.

Value adding software engineering could be defined to be a transformation process (Figure 11) where, based on the need of a customer, a set of requirements are defined and then transformed through different phases to a solution fitting the customers need. Mili et al. (1995) define these to be transformations and/or translations of the description of the desired system from one language to another. Depending on the chosen process model, activities of value adding software engineering can be implemented like waterfall, incremental etc. Figure 11 is an example where the results of value adding activities are defined as: requirements model, design model, software code and resulting system.

Verifying means the activities that are required to assure that the results from the value adding actions are the ones wanted and are of good quality. Actually the word verifying is used in this thesis in a broad sense to include some validation related activities too. Examples of verification activities are inspections/reviews and testing. In Figure 11 the verifying activities are not visible, but those could be e.g. requirements reviews, design reviews, code reviews and different kinds of testing, for example, unit testing and system testing.

In addition to the value adding and verifying activities defined by Kylmäkoski (2006), an activity type is required for managing the work. It is important, especially, because software engineering work is team work requiring coordination. Edwards (2003) indicates that a project requires some higher-level activities: project management, control, and people management. The managing software engineering activity type includes those. Very seldom can a software engineering project be made by one person only. Because of time pressure and competence requirements in projects, several persons are required to achieve project goals. The work is divided into tasks and the tasks are distributed to different persons. This results in the need for knowledge sharing including strong communication and coordination, and requires some kind of managing activity type.

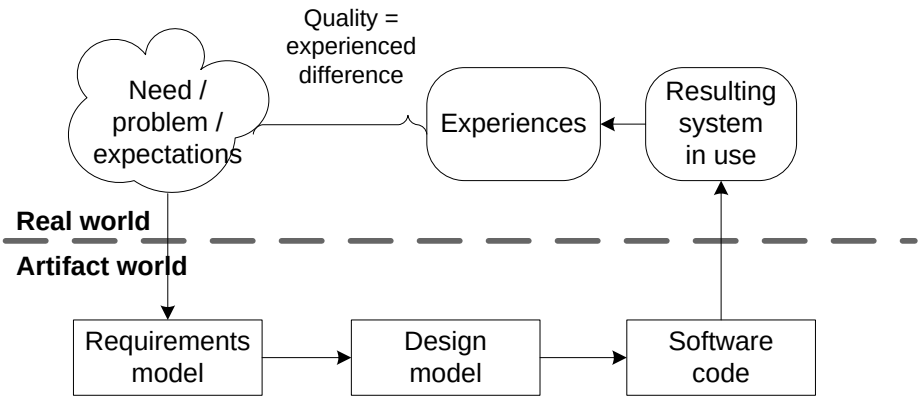



Figure 11: Value Adding Software Engineering is a Knowledge Transformation Process.

Value adding engineering and also verifying engineering are largely based on professional knowledge. Sveiby and Lloyd (1987) point out that in addition to professional knowledge managerial knowledge is needed. The managing activity is based on this managerial knowledge. Managing activity includes activities that continue throughout the software engineering work. Examples of managing activities are project planning, monitoring, and controlling activities.

## 3.3 Project Life Cycle Dimension

In addition to knowledge sharing interface and software engineering activity types, there are other important features of software engineering work to be noticed. In particular, the project life cycle dimension (Figure 12) is essential for understanding knowledge sharing in software development. In this study, a project life cycle means project phases at a very general level, not a certain software development project lifecycle.

From the knowledge sharing perspective, a project has two very critical phases regarding knowledge sharing, the beginning of a project and the closure of a project. In the beginning (here called establishment) it is crucial to make sure that all required knowledge is adequately available for a project in the form of people, documents, requirements etc. At the project closure, the challenge is to assure that knowledge gained in the project is properly shared or stored so that all the parties in the organization who will need it later can have ready access. In order to complete the project life cycle, the project realization phase must be included also. The resulting three phases are establishment, realization and closure (Figure 12).

Practically, the establishment phase means the work required to start a project and to collect the basic set of requirements for it. The project realization is the actual implementation of the project and the project closure is the phase, when the project is officially closed and evaluated.

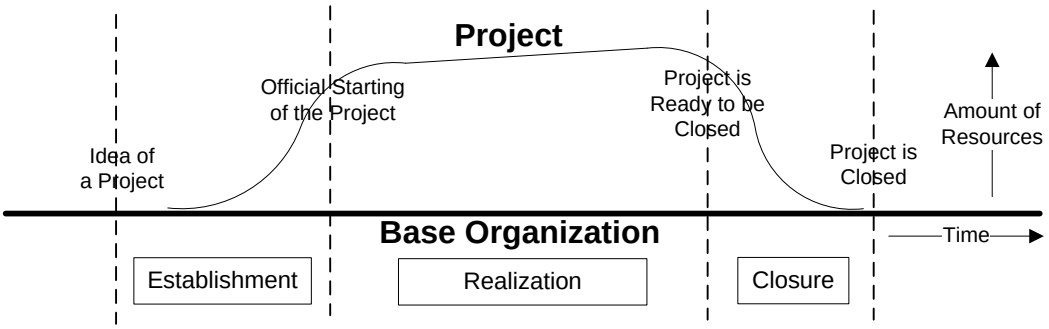



Figure 12: Project Life Cycle.

## 3.4 Knowledge Sharing Framework (KSF)

The purpose of the Knowledge Sharing Framework (KSF) is to identify situations where knowledge sharing should take place. By studying such situations, the knowledge sharing status in certain environments can be made visible and possibilities can be found for improvement.

KSF utilizes the knowledge sharing interfaces, software engineering activity types and project lifecycle phases described in the previous sections. Those are identified here, with the general term, dimensions. A summary of the elements in these dimensions is introduced in Figure 13. The basic part of the KSF consists of dimension I, knowledge sharing interfaces. Those are: knowledge sharing in a project team, in an organization, in the customer and supplier relationship, and in unofficial knowledge sharing. The interface I2 is further divided into knowledge sharing between current projects, from previous to future projects and between a project and the organization where it is undertaken. Dimension S is based on the software engineering activity types: managing, value adding and verifying software engineering. Dimension L then follows the project life cycle as discussed earlier.

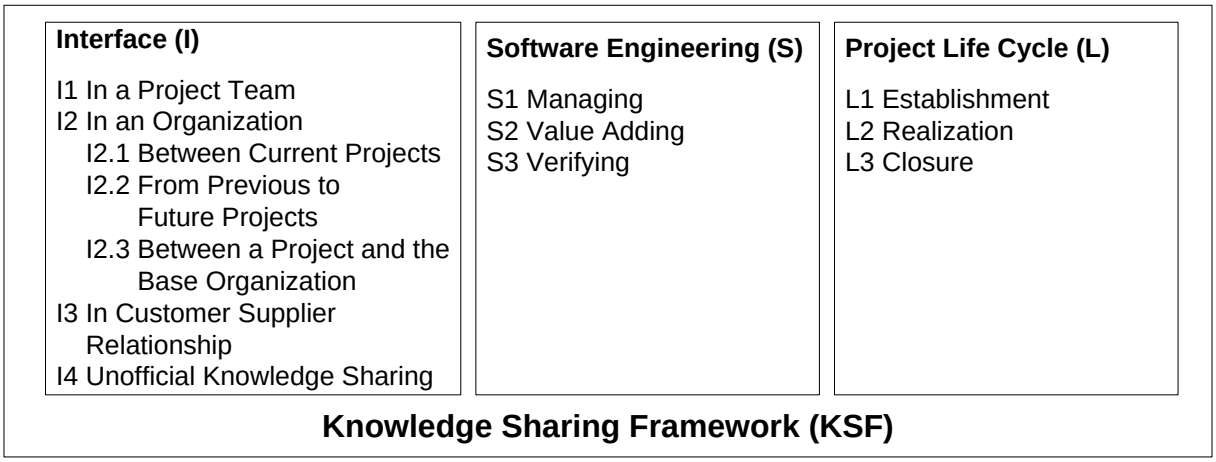



Figure 13. The Knowledge Sharing Framework Dimensions.

It could be discussed whether I4, Unofficial Knowledge Sharing is at the right place at the interface dimension I. It is different from the other interfaces (I1-I3) which are very concrete ones compared to I4. Even knowing this challenge, interface I4 was included, because it proved to be useful viewpoint when utilizing KSF in case studies and other situations reminding about the unofficial knowledge sharing.

KSF is a tool for identifying situations where knowledge sharing should take place. It is used through studying the different combinations of dimensions I, S and L. In combinations each dimension has its own role as described in Figure 14. It is a combination of the interface (where), activity type (what), and project life cycle phase (when). All combinations are valid. In this study, however, the combinations with I4 (unofficial knowledge sharing) are not the focus area because of their informal nature. Those are a very important part of knowledge sharing in organizations, but in this study the focus is in the more formal knowledge sharing interfaces I1-I3.

Through these combinations the software engineering work can be thoroughly covered. What actually is studied are the different situations in the knowledge sharing interfaces. The knowledge sharing actions cannot be separated from the environment where those occur. Thus, it needs to be studied as part of the software engineering work. Software engineering activities in knowledge sharing interfaces are studied from the perspective of a different type of activities (software engineering activity type) and how those differ in various phases of the project life cycle. Next, some of the combinations are briefly introduced to show the richness of this framework.

From the knowledge sharing interface I1, an example combination could be: I1, S2, L1. It refers to *knowledge sharing in a project team* and between the project team members. S2 refers to *value added software engineering* activities and L1 refers to *project establishment*. One example of knowledge sharing involvement is the definition and sharing of requirements at the beginning of a project. To start a project well all project team members need to know what the results of the project are expected to be. Requirements are one way of sharing this kind of knowledge.

For the knowledge sharing interface I2, *knowledge sharing in an organization*, an example combination could be: I2, S1, L2. It refers to *managing software engineering* (S1) during *project realization* (L2) in that interface. For example, this could be the resource management of an organization having several ongoing projects. Resource management requires knowledge sharing between projects, normally meaning the project manager, and the resource managers in the base organization (I2.3). A project needs adequate resources (amount and competences) but the need may vary during the project. At the same time there are several ongoing projects which all need resources. To manage this situation the resource managers need to communicate with all projects, and also to have basic understanding of the project statuses and estimated changes in resource needs.

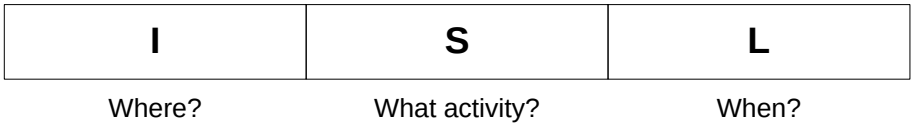



Figure 14. Combination of Dimensions.

The combination I3, S3, L3 refers to the *knowledge sharing interface between the customer and the supplier*. This combination *verifies software engineering* at the *project closure* phase. One example of this would be the knowledge sharing required to have the approval for the project by the customer, including the acceptance testing, coordination & communication related to it. Another example could be the knowledge sharing actions required to understand the customer satisfaction.

The activities for software artifact reuse would take place at interface I2 either between projects (I2.1 or I2.2) or between the base organization and a project (I2.3). An example of the first could be reuse of a software component created in an earlier project (I2.2). Product line-based approach and activities would require knowledge sharing at least with the base organization (owner of the product line assets) and the project utilizing those. This kind of knowledge sharing is mainly an example of value adding knowledge sharing which have slightly different targets in different project life cycle phases.

Combinations with S2 (value adding) and L3 (project closure) would mean a value adding software engineering activity during project closure. That is not normally a wished-for case, but could take place in real life also. During project closure (L3) only managerial closure activities (S1) including lessons learned sessions should continue. An example of a situation where S2 (value adding) and S3 (verifying) activities would still be required could be e.g. delayed testing or even coding continuing still parallel to project closing activities.

# 4 APPLYING KSF

In this chapter KSF is applied to an industrial case study in one telecoms software company. It has also been used as a conceptual analytical framework for studying the potential of knowledge sharing support in three different software development methods. Finally, based on these applications, some conclusions are made about the applicability of the KSF.

## 4.1 Case Study Projects

### 4.1.1 Introduction

In order to systematically collect evidence of knowledge sharing reality in software engineering, an industrial case study was implemented in a Finnish unit of one global telecoms software company. The work in the company is organized as projects which are normally undertaken for external customers. The phases defined by Eisenhardt (1989) were used in this study. The case study was implemented in steps having one problematic project studied first, then one successful project. These projects are referred to as Alpha and Beta. After studying these projects, a web survey was implemented to have information of the general situation in other projects in the company regarding knowledge sharing.

When selecting a project to be studied, one requirement was that it should have full project management responsibilities in the company. Several then ongoing projects were more customers' driven and thus left out. Project Alpha was selected because of a practical need to properly study the unsuccessful project to avoid similar problems later. Because project Alpha was one of the worst projects in that company's history, we wanted to select a successful project to be the other project to be studied. The selection of project Beta was made based on the customer satisfaction survey results, and based on the indicators in internal project reports. Some of the ongoing projects were very new ones and those were left out from this selection because there would have not been enough material and experiences to be studied.

Both projects introduced new technologies for mobile phones and had challenges in defining requirements for the project. The main reason for this was inexperience with the new technologies that had not been commercially launched or supported earlier. Projects were studied toward the end of their life cycles (before the maintenance phase), meaning that there was a large number of different kinds of materials to study. The key people in the selected projects were interviewed. Project Alpha was slightly more complex than project Beta meaning that the projects should not be compared with each other directly. These projects, however, are not totally different from each other and they can be good sources for identifying possible challenges and strengths in knowledge sharing.

During its existence project Alpha involved totally 19 participants, the core team being about 10 persons. After the problems started to be fully visible, there were several changes in the

project team, also in the core project team. The additional nine participants were people that were assigned to the project later to ”get it back on track”. These nine additional persons did not work at the project at the same time. The project team in project Beta was six persons.

The main data collection method has been semi-structured interviews. The interview questions are listed in the Appendix A. Data collection for these two cases were implemented during March 2005 - February 2006. For project Alpha there were four interviews and for project Beta there were two interviews. Three key persons in project Alpha were interviewed as well as one designer being responsible for one key module. Two of the key persons were project managers (at different periods) and one was the head designer of the project. In project Beta the project manager and the head designer were interviewed.

The amount of interviews, six interviews, is not very high compared to the amount of people in those projects. Even though there could have been more results if adding more interviewees, the results already showed some repetitiveness in the findings and confirmed that the most important findings should have been covered. In this case study the target was not to have all possible findings covered, just to have enough findings to prove the usefulness of KSF.

The first interview in project Alpha was a pilot interview where the use of the framework was piloted. After this interview the actual interview questions were finalized and used. The first pilot interview had to be taken much earlier than the others because the interviewee was about to change to another project and the researcher wanted him to be able to remember everything clearly before being reassigned to another project.

The interviews started with general questions designed to find out the successes and problems in the project. After that, the questions were prepared to follow the KSF tool. When defining the questions, based on the three KSF dimensions, a three dimensional matrix was made. For each cell of the matrix, the related software engineering project activities requiring knowledge sharing were identified. Questions were made so that knowledge sharing (or lack of it) related to the activities should be identified. Because the amount of time available for an interview was limited, some cells in the matrix had to be ignored. The result was thirty questions (introduced in Appendix A), some of those covering more than one cell in the matrix. In addition to those questions, there were questions clarifying the project background.

The interviews were recorded and later typed. Possible findings were searched from the typed material and categorized. All possible findings were listed on Excel table and categorized according to the three KSF dimensions. The results were rearranged making possible to study different KSF combinations separately together. When more than one source supported the possible findings, those were defined as findings.

In addition to the interviews, existing project material (project plans, progress reports, final reports etc.) and observations were used to look for further evidence. Validation of results was made through triangulation (confirming a potential finding based on other available sources) and through confirming the findings together with some interviewees and line managers.

Based on the findings from these two case projects, the studied organization initiated several improvement actions. Examples could be full rearrangement of expert support and establishment of Projects Unit in the organization. The first one of these aimed at ensuring the efficient use of the most competent experts, and the latter to support project managers and to have organization level visibility to project results and situation.

## 4.1.2 Project Alpha

### 4.1.2.1 Overview of the Results

Even though project Alpha produced the results required, it confronted several problems during its life time. Altogether 15 problem areas were found, as introduced in Figure 15. Problems have a consequential nature. Some problems could be defined as seed-bearing, in other words, they lead to other problems later. The arrows in Figure 15 describe the consequential relations between the problems. In this study the analysis is limited to the project and its close environment.

Two problems are defined as the main problems in this case. Those are: P1 Weak Project Definition and P2 Resourcing with Inexperienced Resources. Those are selected as main problems, because they are foundational (several other problems result from these) and the other three originating problems (P3-P5) are minor problems compared to these two. Problem P3, Failure in Providing Organizational Support, has an arrow to problem P2. Problem P3, however, is not identified as an originating problem for problem P2 because it is a problem that together with P2 makes the inexperience more visible. Bold arrows in Figure 15 show the main problems and their consequences. There are three main consequence paths that are numbered to make those visible in the figure. These paths are better introduced in Section 4.1.2.2.

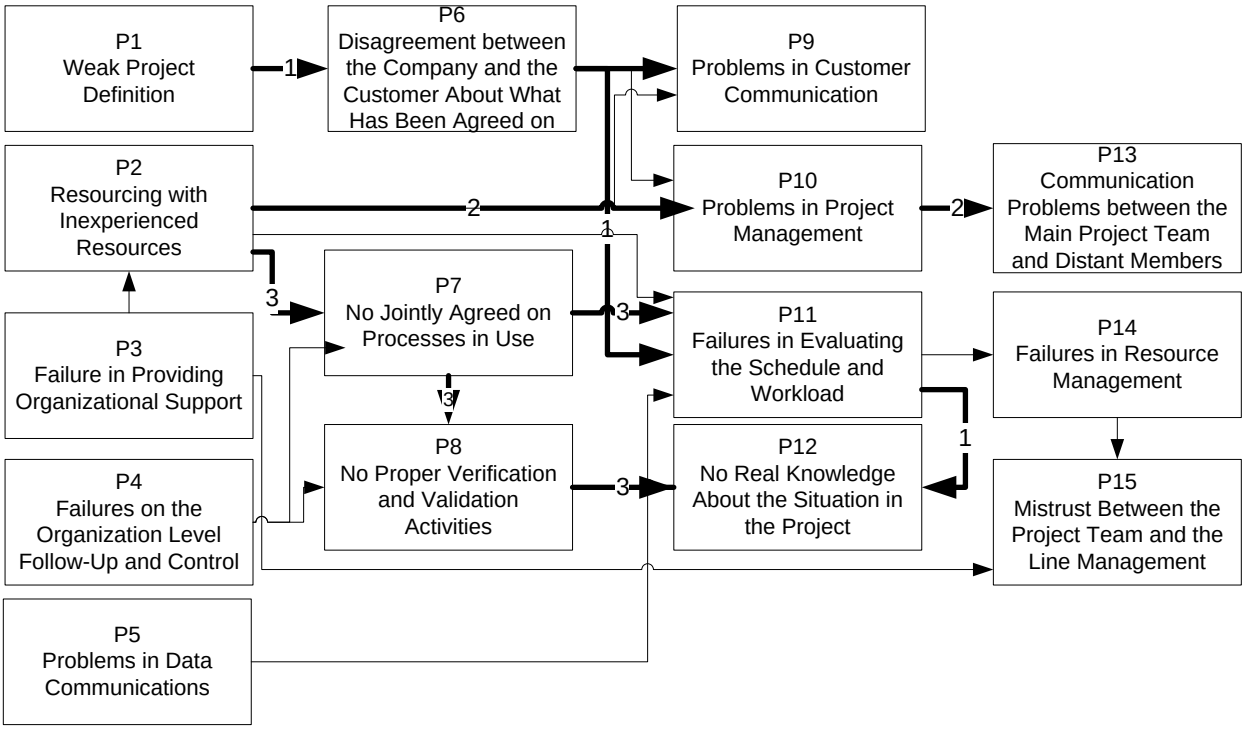



Figure 15. Identified Problem Areas of Project Alpha.

All elements of the KSF were identified in project Alpha. The findings presented in this thesis and several other findings are the result of the use of the framework. The most difficulties were related to the elements I2 (knowledge sharing in organization) and I3 (customer supplier relationship). Among these, the areas of S1 (managing software engineering) and S3 (verifying) were the most critical. Problems in these areas resulted also in problems in the area of I1 (knowledge sharing in project team). The element S2 (value adding software engineering) seemed to work well, producing acceptable results, even though the problems in other elements affected it (delays etc.). The main problems originated at the project establishment phase (element L1) and affected the project throughout. At the closure phase (element L3) most of the problems had been solved and also many preventive actions were initiated to avoid these kinds of problems later.

#### 4.1.2.2 Findings per a KSF Element

I1 element (knowledge sharing in a project team) was implemented in a very informal way. In many situations it was nearer to the type I4 (unofficial knowledge sharing). The project team (excluding the distant members from the other site) felt that they had very good spirit. However, because of not having systematic ways of knowledge sharing, the team members felt unsure.

Knowledge sharing in the organization (I2 element) was unsuccessful. There was a considerable lack of knowledge sharing between the right people and some of the requests for help were not taken seriously or there just were not enough people to help with the project. Problem P2, Resourcing with Inexperienced Resources, could be an example of the findings related to the I2 element. When the project was established, the project team consisted of new recruits in the company. Based on the understanding of the line management there were both experienced people and some less experienced people on the team. The situation was reasonable from the management point of view. The actual problem was, however, that the experience was not exactly the type required in the project. At that time, it was also impossible to release experts from other projects (P3) to support this project. They tried to solve the lack of competences later with several other resources. As a result, there were about 19 different persons somehow involved during the project.

Customer communication (I3 element) started well, but after the misunderstandings appeared between the customer and the company it did not work anymore. As an example of the problems in I3, one of the interviewees described the weak project definition (P1) as “The analysis for requirements specification were not detailed enough but was one that we could live with. What was missing was what the project itself should be.” This meant, in practice, that the required quality level of the results, the acceptance criteria, was not stated clearly enough. The project definition should have been the result of the common understanding between the customer and the company. The parties, however, had different understanding about the desired results. The weak or very general project definition covered the understanding of both parties, because it did not identify those parts where the differences were.

Problem P1 (weak project definition) resulted in several other problems during the project (see path 1 in Figure 15). First, it resulted in the disagreement between the customer and the company about what had been agreed on (P6). This, in consequence, resulted in e.g. problems in

customer communication (P9) and failures in evaluating the schedule and work load (P11). Finally it meant that there was no real knowledge about the situation in the project (P12).

When using the S dimension of the KSF, the S1 element (knowledge sharing in managing software engineering) did not exist in a systematic way. S2 (value adding software engineering), however, worked quite well by ignoring the lack of systematic routines and the delays. Actually, the customer has even praised the architectural solutions. The last element S3 (verifying software engineering) did not exist in a systematic way.

Because of inexperience, there were several problems in the project management (P10, see also path 2 in Figure 15). In the beginning there were no systematic enough ways to define and divide tasks, to follow the project progress etc. Also three of the four interviewees said that everybody knew that things are going wrong but nobody really had the courage to raise this issue. This was not only at the project team level but also at the organization level. The main reason given for not interfering was that the project was "two weeks from ready" for a long time during its life time.

Part of the project team was on another site. Because the project was not coordinated well enough (P10), it was very difficult to manage the resources (two engineers) that were at the distant site of the project (P13). This dearth of cooperation between sites was strongly criticized on both sides and lack of many kinds of knowledge sharing between the persons on the main site and the distant site was evident.

In the beginning of the project, mostly, because of the inexperience (P2) but also partly because of the organizational follow-up failures (P4), the team and especially the project manager did not really understand the purpose of commonly agreed processes or work flows (see path 3 in Figure 15). The project manager said "Actually processes make me feel sick. It suits others better to do everything according to the predefined instructions." Not following the predefined processes the project had a lack of systematic project planning, monitoring (P11), verification and validation activities (P8), which are an example of element S3 (verifying software engineering). Of course, some reviews and testing existed in the project but those were not very well planned, scheduled nor resourced. Again, these all resulted in a lack of really knowing the situation in the project (P12). Problem P8 (No Proper Verification and Validation Activities) meant that with some persons there were no ways to assure that right things were done promptly and that results were usable. There were, for example, situations where a person was leaving the project and the results achieved proved to be far from what was expected. The results became visible to the other team members only when they had to continue the work based on the results of the team member who was leaving.

### 4.1.3 Project Beta

Project Beta was a very successful project. The main problems were the defects in the development platform hardware and the availability of software components produced under third party intellectual property rights. The corrective and preventive actions for both problems actually promoted more knowledge sharing. The hardware problem initiated more communication between the customer and the project manager. The availability problem resulted

in a new checkpoint on the project checklist which was subsequently used in all projects. This is an example of I2, sharing knowledge in organization, and especially I2.2, from previous to future projects.

In the project Beta knowledge sharing in the project team (I1 element) seemed to be well implemented. The interviewees thought the weekly formal project meetings (example of S1, managing software engineering, element) were very useful and the task pool system that was used in assigning tasks worked nicely. The tasks in the task pool were planned together in the team and managed by the project manager. During the weekly meetings the situation with the current task was checked and new tasks assigned when needed. Unofficial knowledge sharing (I4) was encouraged, by having all the team members in one room together with another team including the expert named for this project.

Knowledge sharing in the organization (I2 element) also seemed to work well in this project. A nominated expert was available when required and who also sat in the same room as the project team. One line manager had been named as a Project Director and he was very active in this role. A Quality Assurance Coordinator was named for the project to support the project manager in quality assurance related topics. These are examples of I2.3 element, knowledge sharing between a project and the base organization. The naming of a Quality Assurance Coordinator was actually one result of the preventive actions initiated based on project Alpha and thus an example of knowledge sharing from previous to future projects (I2.2).

I2.1 (knowledge sharing between current projects) was visible in the contacts with at least two other project teams to learn more about their experiences among certain technology areas. I2.2 element, knowledge sharing from previous to future projects, was visible in that the project was fully based on the previous project and there were also possibilities to have future projects based on the current one. The resulting designs, code etc. from this project could not be directly utilized in other projects, but the experiences and newly-piloted ways of working were possible to share.

Knowledge sharing between the customer and the company (element I3) worked very well. This relationship was particularly open and strong. The difficult issues were quite easy to discuss between the customer and the company. The communications included the managing element (S1) in steering group meetings and the value adding element (S2) in weekly technical meetings. The verifying element (S3) came from the joint reviews.

The interview of one software engineer in project Beta produced the understanding of one set of differences between projects in general from the knowledge sharing perspective. It was related to the element I3, knowledge sharing in customer supplier relationship. This interviewee had been used to projects where nearly all members of the project team participated in the customer communication. In this project, however, the communications model was project-manager centric, the project manager being one of the very few persons in the project team who talked directly with the customer.

In project Beta, a question, common with the project Alpha, emerged: how to know that a software engineer producing code does progress as expected and does produce results of the required quality? This is one place where more knowledge is required, whether it is knowledge that a certain engineer can be trusted (tacit knowledge based on experience) or to have knowledge of ways in which to check the situation in time, before the opportunities for

corrective actions are lost. In project Alpha this question was the result of not having systematic enough verification and validation activities (problem in element S3 verifying). In project Beta this was not the case. Project Beta had quite reasonable and systematic verification and validation procedures in use. Those, however, were not always enough to follow the progress of single project team members (element S1 managing software engineering).

The project life cycle dimension L was mostly visible in project Beta in the successful introduction of the project to the project team members (L1 project establishment). Project Beta was preceded with another project where also the phase L3 (project closure) included well established actions, e.g. lessons learned collection, but here the study was restricted only to project Beta.

### 4.1.4 Projects Snapshot

#### 4.1.4.1 Overview

To get better general understanding of the knowledge sharing situation in the studied organization a survey using web-based questionnaire was implemented after the more thorough cases Alpha and Beta. A questionnaire was made based on the KSF. It was targeted to the then ongoing projects in the organization and it was sent to the project managers. Pure resource-hiring types of assignments were not included. Project managers of eight of the 12 then ongoing software engineering projects answered the survey.

One of the four project managers who did not answer to the survey told that he is too busy to answer. The other three did not give any explanations. Based on the customer satisfaction survey results and internal project reports none of these four projects were trouble projects. The customer satisfaction survey results and other indicators showed that those were either average or well-performing projects.

The sizes of the project teams varied from three to over forty persons per project thus making it unreasonable to compare the projects with each other. The purpose of the survey was to have organizational-level understanding of potential common-knowledge-sharing best practices and problem areas in the organization.

The questionnaire included background information (e.g. name, project identification), open questions about the project in general (e.g. What has gone well in the project? and What has not gone so well?), a question about How the project has/most probably will achieve its targets? and two tables of multiple choice questions asking for the success/problems in project establishment (Table 3) and project realization (Table 4). Because the projects were ongoing projects, questions about the project closure phase were not asked.

This approach was piloted to find a way to quickly understand the situation of an organization unit related to knowledge sharing. Studying all existing projects, the way done in the cases Alpha and Beta, would require too much time to be easy and quick to implement.

#### 4.1.4.2 Findings

Based on the results all eight project managers saw that their project had achieved or probably will achieve the targets of the respective project. The scale used for this question was from poorly (1) to excellently (5). The result was good (4) for all projects. Open questions were used to ask what had gone well (potential best practices) and what had not gone so well (problems) in the project. Six project managers of eight told that the customer interface (I3) had worked very well and that problem solving over this interface worked well. One project manager reported that when increasing the amount of meetings with the customer to two per week the communication started to work very well. Five project managers of eight reported that knowledge sharing in the project team (I1) worked very well including also good team spirit and strong motivation increased among the project team members. Very often the main reason for this seemed to be a very interesting project and also the strong competences of the project team members.

The problems or challenges in the projects were not very similar between projects in the same way the successful things were. Positively, project managers did not see large problems in projects. Some single problems were mentioned per project. Most of those were knowledge-sharing related problems such as change requests coming very late from the customer. In addition some problems were caused by the hardware used.

The project managers were asked to report if they had observed any reoccurring problems within projects in the organizational unit. One of these problems was related to the organization structure, having several offices and making knowledge sharing difficult. Other comments were related to effort estimations and very minimal knowledge sharing between projects and not knowing who really knows what in the organization.

In addition to open questions, the project managers were asked to rate given statements based on how successful or problematic those had been in the project. The used scale was: 1 *success*, 2 *not problem*, 3 *problematic* and 4 *extremely problematic*. In addition, selection *not valid in this project* was available. The statements and the amount of the answers are given in Table 3 and Table 4. The tables include also reference to the KSF combinations after each item. For example, the first line in Table 3 is created based on the KSF combination I2, S1, L1. The statements at Table 3 are related to the project establishment (L1) and the statements in Table 4 to project realization (L2).

The statements in tables Table 3 and Table 4 are in the order they were asked from the project managers. During the project establishment (L1, Table 3) two project managers reported that the definition and agreement of requirements for the project with the customer were problematic. One of them explained “Requirements were not clear at the beginning to anyone.” In this case, the project was by nature a research project, where there were constant changes to the requirements. Similarly, some problems were perceived by two project managers regarding to having shared understanding with the customer.

Table 3. Project Establishment (L1). KSF elements reference in parenthesis after each item.

| Item | Success | Not problem | Problematic | Extremely problematic | Not valid |
|---|---|---|---|---|---|
| The transfer from sales to project implementation was implemented successfully. Project manager had enough knowledge to start the project. (I2, S1) | 1 | 4 | 0 | 0 | 3 |
| When project started, shared and detail enough understanding existed between the customer and the project about the project and its objectives. (I3, S2) | 4 | 1 | 2 | 0 | 1 |
| In the beginning of the project requirements were documented, agreed with the customer and this set of requirements has been used as the baseline in change management. (I1 & I3, S2) | 1 | 3 | 2 | 0 | 2 |
| It was easy to understand what the objectives of the project were. (I1, S2) | 4 | 2 | 1 | 0 | 1 |
| The project has a named and active Project Director and QAC. (I2, S1 & S3) | 1 | 3 | 0 | 0 | 4 |
| Project work was divided into reasonable tasks. (I1, S1) | 4 | 2 | 1 | 0 | 1 |
| Experts were named for the project and they were available when required. (I2, S1) | 3 | 2 | 0 | 1 | 2 |

As can be seen from Table 3, potential best practices might be found regarding to having common understanding with the customer about the results of a project (4 success ratings), easy to understand project objectives (4 successes), and division of project work (4 successes). It must be noted that these were the project managers evaluating their own projects. Asking other team members might give different results.

During the project realization (L2, Table 4), four projects had challenges in having proper updated project plans (2 problematic ratings and 2 extremely problematic ratings). Also the gathering of lessons learned was problematic based on the answers. Four project managers defined it problematic. Another challenge was the efficiency of reviews. There were two project managers who stated that to be problematic. In addition, efficiency of testing was also found problematic by two projects managers answering to that question. Most of these problematic areas were related to the S3, verifying software engineering, so it seems to be a candidate for improvement.

Knowledge sharing with sub-suppliers (supplier's suppliers) might have potential best practices. Three project managers answered this statement and they all reported that it had been a success. Only one project used suppliers directly. . In two other projects, the customer had also additional suppliers and this answer refers to the cooperation with them.

During the project realization (L2) potential best practices might be found in customer communication (5 successes), division of tasks, expectations from the project, common understanding with the customer about the deliverables, and customer satisfaction to the project (4 success ratings for each of these).

Table 4. Project Realization (L2). KSF elements reference in parenthesis after each item.

| Item | Success | Not problem | Problematic | Extremely problematic | Not valid |
|---|---|---|---|---|---|
| The project has had a proper and updated project plan during the project. (I1, S1) | 2 | 5 | 2 | 2 | 1 |
| Project team members are committed to the project's schedule and believe it can be implemented. (I1, S1) | 3 | 3 | 1 | 0 | 1 |
| Division of tasks/assignments to team members has worked well. (I1, S1) | 4 | 3 | 1 | 0 | 0 |
| Changes have been well managed and communicated with the customer (I1 & I3, S1) | 2 | 6 | 0 | 0 | 0 |
| Changes have been well managed and communicated in the project team (I1, S1). | 3 | 4 | 0 | 0 | 0 |
| Requirements traceability exists in such a level that it helps evaluating effects of changes. (I1, S2) | 1 | 4 | 1 | 0 | 2 |
| It is easy to understand what is waited from you in the project. (I1, S2) | 4 | 4 | 0 | 0 | 0 |
| Effective and well planned reviews etc. were implemented during the project. (I1, S3) | 1 | 2 | 2 | 0 | 3 |
| Effective and well planned testing was implemented during the project. (I1, S3) | 1 | 3 | 2 | 0 | 2 |
| Customer communication functioned well. (I3, S1-S3) | 5 | 2 | 1 | 0 | 0 |
| It has been easy to have acceptance from the customer for different project results and deliverables. (I3, S1) | 3 | 4 | 1 | 0 | 1 |
| Knowledge sharing (communicating, information & experience sharing etc.) worked well in the project team. (I1, S1-S3) | 3 | 5 | 0 | 0 | 0 |
| Knowledge sharing worked well with the remote participants of the project team (people in other offices). (I1, S1-S3) | 2 | 3 | 0 | 0 | 3 |
| Knowledge sharing worked well with the suppliers. (I3, S1-S3) | 3 | 0 | 0 | 0 | 4 |
| Knowledge sharing with all interrelated projects works ok. E.g. with projects that produce results to be used by this project or vice versa. (I2.1, S2) | 1 | 4 | 1 | 0 | 1 |
| This project utilized beneficially results from earlier projects. (I2.2, S2) | 3 | 4 | 1 | 0 | 0 |
| Project has had available reasonable resources (amount and competences). (I2, S1 & S2) | 1 | 5 | 2 | 0 | 0 |
| The project and the customer have the same understanding about the project deliverables and schedules. (I3, S2) | 4 | 4 | 0 | 0 | 0 |
| The project uses well defined processes and every team member knows which processes are used. (I1 & I2, S3) | 1 | 5 | 2 | 0 | 0 |
| Project manager can easily follow the progress of the project compared to plan. (I1, S1) | 1 | 4 | 2 | 0 | 1 |
| Project manager can easily follow the progress of single team members. (I1, S1) | 1 | 6 | 1 | 0 | 0 |
| Project is in schedule and has produced the results required by now. (I1, S1) | 3 | 4 | 0 | 1 | 0 |
| The customer is satisfied to the project and its results by now. (I3, S1 & S2) | 4 | 4 | 0 | 0 | 0 |
| Lessons learned have been systematically gathered during the project. (I1, S3) | 0 | 4 | 4 | 0 | 0 |

The survey resulted in a basic understanding of the situation with knowledge sharing in the organizational unit at certain times. The results, however, were a bit too general to initiate improvement actions in the unit. Only one potential area was found. This was the verifying software engineering (S3) during the project realization (L2). The results from this project survey are only suggestive.

### 4.1.5 KSF Profiles

A KSF profile is a visual way of presenting results from a knowledge sharing survey based on the KSF. It summarizes the results and gives quick understanding of the general situation. The cases of Alpha and Beta, together with the KSF, made knowledge sharing and also the lack of knowledge sharing very visible. Matching of the KSF with the findings, the KSF profiles are introduced in Figure 16. The knowledge-sharing interface (I) elements are shown as columns and the lifecycle (L) elements as rows that are further divided into sub-rows for software engineering elements (S). The changes between lifecycle (L) element rows indicate changes over time.

The combination S1 (managing software engineering) and I1 (knowledge sharing in project team) was problematic in the beginning of project Alpha and mostly successful in project Beta. In project Alpha the challenge was very informal project and task management in the project team. At the end of the project, there was a different project manager and the style had been changed. Thus the combination I1-S1 was not more problematic at L3 (project closure). In project Beta, there was well-managed project team with a respected task-pool system for task division among the project team members. The combination I1,S1,L2 of the KSF profile of project Beta (Figure 16) has both success and problematic elements. The problematic element was the surprise that one team member was not progressing as he should be and this knowledge was gained later than it could have been gained. Similarly, all the combinations of the KSF profiles (Figure 16) can be explained based on the findings introduced in Section 4.1.2.2.

As can be seen, the resulting profiles are very different, reflecting the different level of success in projects. The identified problem areas indicate possible improvement areas, especially if those are visible in other projects also. Identified successes then are potential patterns that should be studied and shared with other projects. For example, many of the good practices of project Beta could have really much helped project Alpha. In project Beta, some preventive measures were taken based on the first lessons from project Alpha.

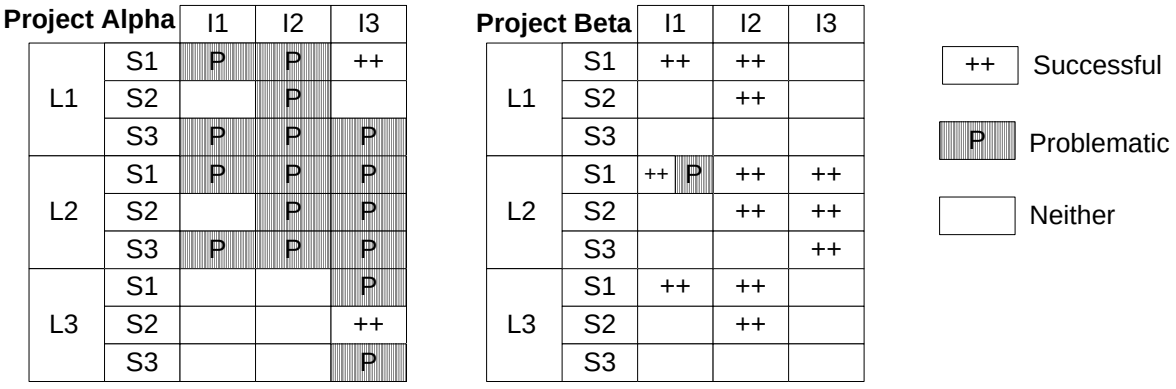



Figure 16. KSF Profiles from Cases Alpha and Beta.

Only very suggestive KSF profile can be drawn based on the web survey results because the questions are on very general level. The resulting KSF profile is introduced in Figure 17. Project closure phase (L3) is not introduced there because all studied projects were in project realization phase (L2). As a result from the open question 'What has gone well..', nearly all projects identified several strengths in interfaces I1 (in project team) and I3 (customer-supplier). The reported problem areas were more specific for one project per time. One more common problem area, however, was found in four projects. It was the collection of lessons learned during project implementation (combination I2,S3,L2). Based on the answers to the open question 'What has not gone so well….' the work estimations (I1,S1, L1) could be one potential improvement area even though it seemed also to be a potential strength of some other projects. Efficiency of reviews and testing were problematic for some of the projects (I1, S3, L2).

Especially in the case of the web survey, but partially also regarding to projects Alpha and Beta, the *Neither* classification in the KSF profiles could, in some cases, mean also that the area was not properly studied to classify it. This needs to be noted when reading the KSF profiles.

KSF profiles can be exploited in an organization in order to understand the level of knowledge sharing practices in projects. Project specific profiles can be further summarized into organizational profiles which give an overview of potential improvement targets and strengths in the organization.

The KSF is mostly used in this study as a descriptive framework making knowledge sharing visible. In addition, it has a normative value. The KSF defines situations where knowledge sharing should exist and thus can be used to find the possible lack of knowledge sharing. The KSF profiles nicely highlight these potential knowledge sharing holes as problematic areas.

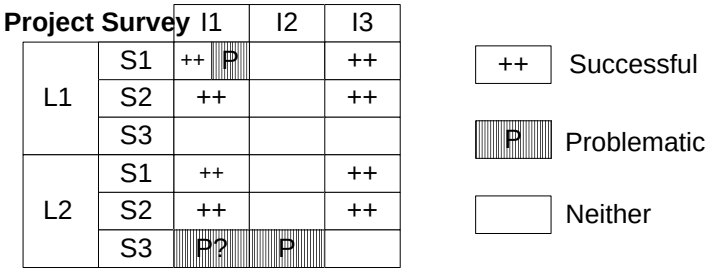



Figure 17. KSF Profile from the Web Survey.

## 4.2 Software Development Methods and Knowledge Sharing Approaches

In this section, the KSF is utilized as a conceptual analytical framework for studying three different software development methods. The object is to understand what kind of environment/preconditions for knowledge sharing these software development methods would create if applied according to the method descriptions. The purpose is not to compare the overall goodness of these methods. Selected methods are studied based on literature, not based on actual experience gained from the use of these methods. It must be noted that in actual use the methods are always some way applied and then the weaknesses found here might be partially reduced with some compensating actions in addition to the actions defined in methods.

Examples of three mainstream software development approaches: traditional, agile and product-line based development are selected as targets. These software development approaches are the current mainstream approaches to software development. The traditional, or plan-driven approach, has its origins in waterfall-based (Royce 1970) thinking. Agile and product-line based approaches have been defined after that. Agile has its origin in the problems of traditional software development approach. The software product-line approach is based on the idea of systematic reuse. The environment for knowledge sharing is very different in these three approaches making those good candidates for studying knowledge sharing in software development.

Used examples of respective methods are OMT++ (Jaaksi et al. 1999) for traditional, Extreme Programming (XP, Beck 2000) for agile, and FAST (Weiss and Lai 1999) for product-line based software development. The results of studying these methods cannot directly be generalized to represent the software engineering approach they have been selected from. These methods are, however, representative to their respective approach, fairly well defined, and used in practice. All selected methods have their place in different kinds of software engineering environments, and their applicability differs in different situations.

Different software development methods or process models are not coherent entities but rather collections of activities following general principles of some software development approach. Different software development methods are not totally interchangeable with each other. It means that the methods could cover partially different area of software development work. The contribution of this part of the research is the understanding of the differences of the methods from the perspective of their potential to encourage knowledge sharing.

Sections 4.2.1, 4.2.2, and 4.2.3 introduce the selected methods and give the analysis utilizing the KSF. In these sections, first, the software engineering activity elements S1-S3 are defined at a very general level, and then knowledge sharing in defined interfaces is introduced. The unofficial knowledge sharing (I4) is covered only at a very general level, because of its unofficial and unpredictable nature. Finally, the actions including knowledge sharing related to the project life cycle phases are briefly introduced. After analyzing the methods separately, a summary is made in the form of KSF profiles in Section 4.2.4.

## 4.2.1 OMT++

### 4.2.1.1 Method Overview

OMT++ (Jaaksi et al. 1999) has initially been extended from the Object Modeling Technique (OMT, Rumbaugh et al. 1991) but later it has been influenced from many other sources. The main phases of OMT++ are described in Figure 18. It is an object-oriented, plan-driven method producing several documents in addition to the final code.

As can be seen in Figure 18, there are two paths, static (upper path in Figure 18) and functional (lower path). The activities on the static path are intended to define and implement the structure of the required software. The functional path aims at defining and implementing the collaboration of the static path entities. When developing more than one application for the same domain the method recommends defining a domain model. A domain model defines the concepts of the domain and the terminology used.

The OMT++ method does not contain many references to the managerial part of software engineering. General recommendations only are given. Jaaksi et al. (1999, p.99) state that when the project team grows the individual productivity decreases dramatically. To help avoiding individual productivity decrease, an individual needs a *sandbox*, an environment where she can work alone, but, with well-defined dependencies to the sandboxes of others. The work of individuals, especially in bigger projects, shall be guided with a controlled process and architectural models. Processes and architectural models serve as a base for project metrics, project management decisions, quality control checkpoints, and scheduling (Jaaksi et al. 1999, p. 104).

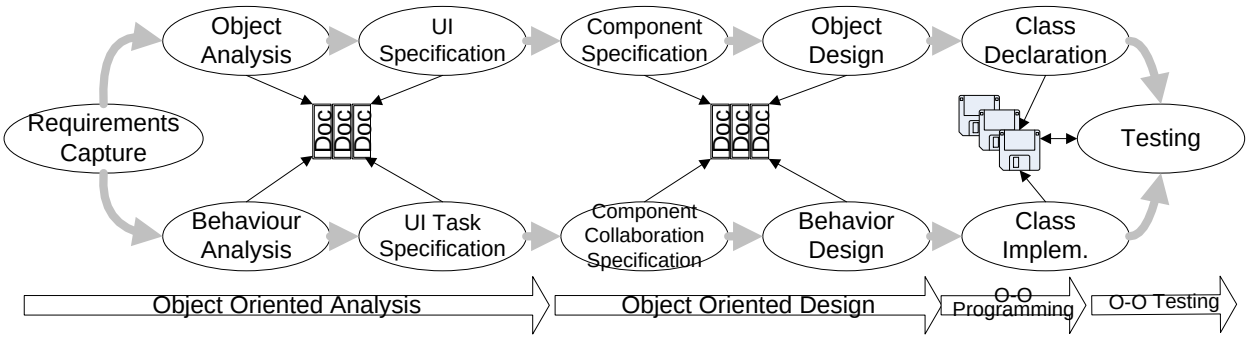



Figure 18. The Main Phases of OMT++ (Jaaksi et al. 1999, p. 7).

#### 4.2.1.2 Analysis with the KSF

Mostly the phases of OMT++ are related to S2, value adding software engineering. The only exception is the phase Testing representing S3, verifying software engineering. Quality control checkpoints are referred to and those represent also S3. Managing element S1 is not very visible in this method but some references e.g. to the important role of controlled processes and architecture models could be thought to represent this view to software engineering. The sandboxes, ways to organize work of individual project team members, belong also to S1 (managing software engineering).

The I1 interface, knowledge sharing in project team, is very much based on the documents and the code, as can be seen from the Figure 18. In practice, the project team members communicate together also using many other ways of communication although the method does not really much highlight those ways. Jaaksi et al. (1999, p. 37) reports: "Documents contain the blueprints of software development. A group of people cannot develop software without proper documentation." Jaaksi et al. (1999, p. 37) finds three purposes for the documents: 1) those provide a visible view to the system in each phase, 2) the documents can be used to document the system and requirements for a later study and 3) the documents form the basis for testing.

The S2, value adding software engineering and the S3, verifying software engineering are mostly implemented in the project team including also some knowledge sharing with the customer (interface I3). The customer relationship and participation of customer representatives is not very visible in the model, but are present in analysis and testing. One exception to this is the introduction of the use cases as an important tool for sharing knowledge between the engineers and the customer (Jaaksi et al. 1999, pp. 12-13). Use cases are used for planning some of the most critical parts of the requirements together with the customer.

The interface inside the organization, I2, is visible e.g. in the references to the domain model. The domain model can be a basis for knowledge sharing between current projects (I2.1) and from previous to future projects (I2.2). This knowledge sharing is based on an artifact (e.g. document) defining the domain model. The method does not have direct references to knowledge sharing between the project and the base organization.

I4, unofficial knowledge sharing, means all kinds of unofficial and informal communication between people. OMT++ emphasizes the importance of sandboxes and documentation. These approaches could be thought to result in the isolation of the work of single project team members and thus might not be encouraging personal contacts. Based on this understanding, OMT++ provides only weak support for unofficial knowledge sharing.

The Requirements Capture phase represents in OMT++ the L1, project establishment. All other phases represent L2, project realization. Therefore, actions requiring knowledge sharing exist related to these two but L3, project closure is not included in this method.

## 4.2.2 XP

### 4.2.2.1 Method Overview

Extreme Programming (XP) (Beck 2000) is used in this study as an example of the agile approach. Abrahamsson et al. (2002, pp. 19) have defined a process model of the life cycle of the XP process (Figure 19) based on the phases defined by Beck (2000). Beck (2000, pp. 131-133) defines that during the exploration phase, the features etc. required by the customer are documented as story cards. This phase is implemented as long as the customer is confident that there are enough story cards to implement a good first release. During this phase there might also be technology explorations and pilots for architectural ideas to have the certainty how to establish a system so that the programmers can make reliable estimates of the effort required.

In the iterations to the first release phase (Beck 2000, pp. 133-134) the schedule is broken into one to four week iterations. For each of those the stories to be implemented are selected based on the customer's discretion. Iteration results in a set of functional test cases for each of the stories scheduled in the iteration. After the implementation of a story, the customer runs the functional tests and the iteration ends in the successful completion of those tests. The architecture is put in place during the first of the iterations.

After the last iteration the productionizing phase (Beck 2000, pp. 134-137) will start. The iteration cycle normally shortens, but the pace at which the software evolves slows down. This phase includes some extra testing, checking and releasing to the customer. After releasing, the maintenance phase starts. It includes maintaining the software and adding some new features in small releases (using the planning and iteration described earlier). When no new stories are invented and the customer is satisfied with the system, it is time for the death phase including e.g. briefly documenting the system.

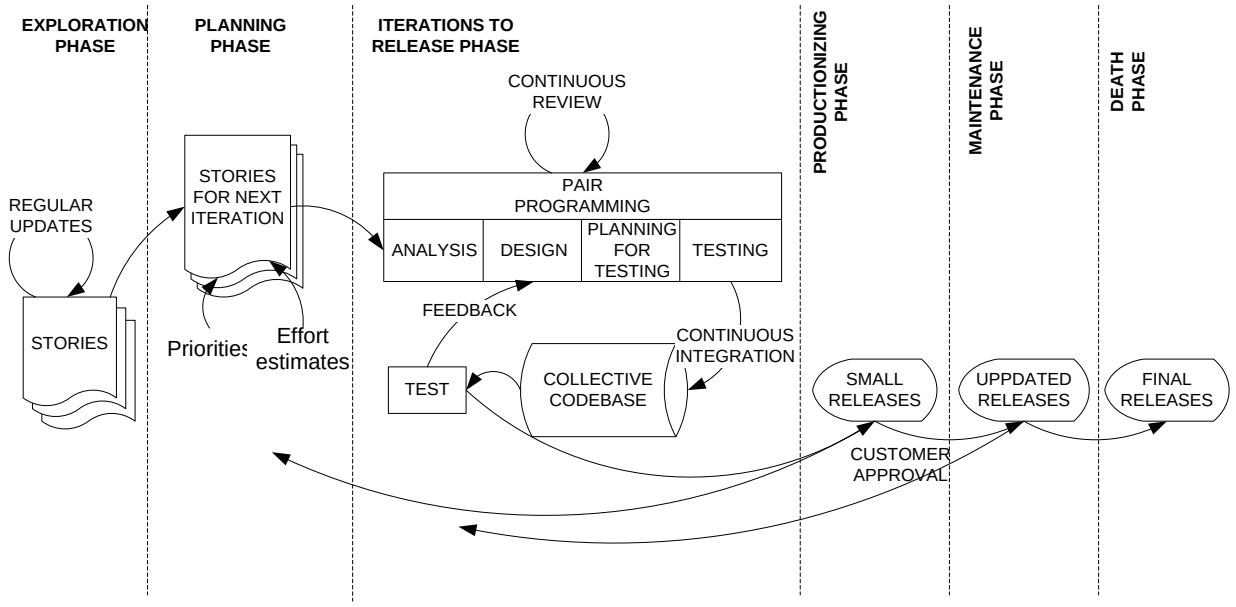



Figure 19. Life Cycle of the XP Process (reproduced from Abrahamsson et al. 2002, pp. 19)

The production code is written with two programmers at one machine (Beck 2000, pp. 54-61). In practice this means that code review is a constant action made in pairs. Another important feature of XP is that before coding, unit tests are made and run (preferably automatically) after writing the code. The customer writes the functional tests which demonstrate that the features are finished. A full-time, on-site customer is continuously available as a participant on a project team.

The manager's job in XP is to run the Planning Game[4] (Beck 2000, p. 74), to collect metrics, to make sure metrics are utilized and to intervene only when a situation cannot be resolved in a distributed way by the team. The manager is more like a coach and the team more like a self-steering team without the traditional authoritative project manager. The communication of system structure and intent is based on oral communication, tests and source code (Beck 2000, p. xvii).

#### 4.2.2.2 Analysis with the KSF

In XP, the value adding element (S2) of software engineering is visible in all phases except in the planning phase. Story cards (the requirements for the project), unit test definitions (could be partially compared to design), and the resulting code represent results from value adding software engineering. The verifying element (S3) then is present in testing and in the pair programming, which actually means a continuous code review. Managing the software engineering element (S1) is implemented in quite a de-centralized way, in the coaching mode. There is more reliance on a self-steering project team.

As Beck (2000, pp. 29-30) writes, "Problems with projects can invariably be traced back to somebody not talking to somebody else about something important". This sentence describes well the point of view of agile methods and especially XP to software development. Direct communication is the most important way of communication. In addition to that, tests and source code are used as tools for communication. These ways of communication apply especially to knowledge sharing interfaces I1 (in project team) and I3 (with the customer). In the I1 interface strong communication and knowledge sharing include all software engineering activity types (S elements).

Through supporting direct oral communication XP could be thought to support also the informal knowledge sharing (I4). At least, making people communicate more with each other could add to the amount of unofficial communication.

The guidance from the customer (I3) is strong through defining the implementation order of the story cards, and by having an on-site-customer in the project team. This guidance relates to the managing element (S1) of software engineering. The on-site-customer then participates in all of the S-elements. The participation of the customer is especially strong through the definition and running of functional test cases (S3, verifying software engineering element).

So, interfaces I1 (knowledge sharing in project team) and I3 (with customer) seem to be very active knowledge sharing interfaces. I2 (knowledge sharing in organization), however, seems not

[4] Quick determination of the scope of the next release by combining business priorities and technical estimates (Beck, 2000, p. 54).

to be noticed much, or being nearly ignored. Actually Beck (2000, pp. 38) says that "today we need to do good work solving today's problems and trust our ability to add complexity in the future when needed". This could be counted as a counterargument e.g. to the reusing based approach of product line based development. XP is not supporting any formal or systematic domain grounding but it comes (if it comes) through the presence of the on-site customer and based on the earlier experiences of the project team members.

The first phase of XP life cycle, the exploration phase, represents the project life cycle element L1, project establishment. The death phase represents the element L3, project closure. The phases in between these represent the L2, project realization.

## 4.2.3 FAST

### 4.2.3.1 Method Overview

A software product line is, according to Clements and Northrop (2002, p. 5): "A set of software-intensive systems sharing a common managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

Software product line could be defined as a production system for software products or a product family based on the core assets. Bosch (2000, pp. 19) says that architecture is the key element to create a paradigm shift from traditional to product line approach. In general, a flexible architecture is a key element of a product line.

FAST method, Family-Oriented Abstraction, Specification and Translation Process (Weiss and Lai 1999) is selected in this study as a representative of product line based software development. It includes three sub-processes: identifying families worthy of investment (qualifying the domain), investing in facilities for producing family members (engineering the domain), and using those facilities to produce family members rapidly (engineering applications) (Weiss and Lai 1999, p. 43). These can be seen in Figure 20. These are further defined with artifacts and production activities in addition to the definitions of organizational roles in the whole production.

After deciding that a domain is economically viable, the domain engineering begins. Its purpose is to make it possible to generate members of a family (Weiss and Lai 1999, p. 53), and it results in the application engineering environment. The environment for application engineering includes an application modeling language to describe the family, tools for creating family members and an application engineering process for using the environment to create domain members (Weiss and Lai 1999, pp. 14-15).

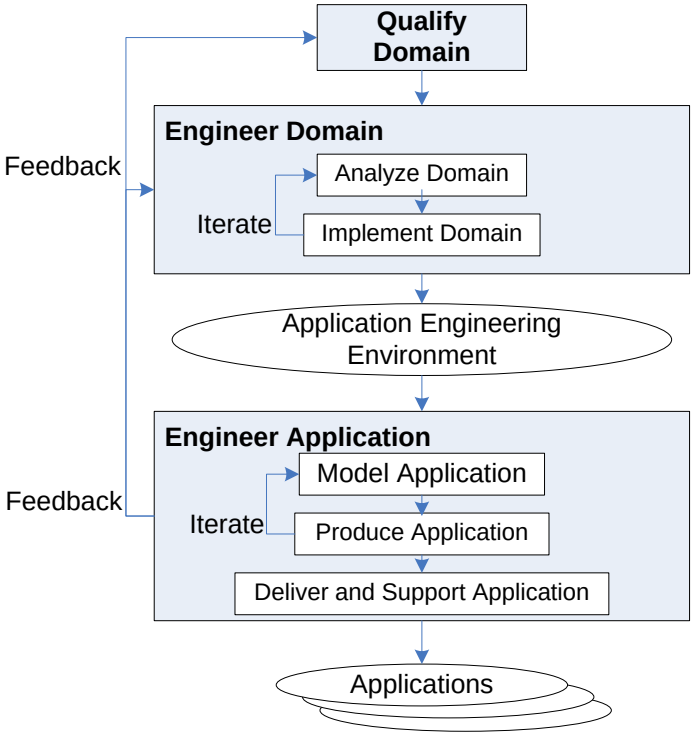



Figure 20. The FAST Process Pattern (Weiss and Lai 1999, p. 44).

In FAST, the production of family members, applications, is very strictly guided. The purpose of the application engineering is to explore the space of requirements for an application quickly and to generate the application (Weiss and Lai 1999, p. 49).

An important part of domain engineering is to create a process model for producing an application. Weiss and Lai (1999, p. 144-145) have introduced the PASTA model as a way to "model complex processes in a graphical, systematic, precise and structured way." They define the PASTA model to be a communications medium. The resulting process model serves different purposes to different stakeholders. For process designers it is a way to represent the process. For environment developers the model serves as a specification for the environment they must develop, and for the software developers it serves as a guide for what they must do at each step in producing the application.

#### 4.2.3.2 Analysis with the KSF

Most of the activities described are related to S2, value adding software engineering element. The S3, verifying element is present in e.g. qualifying the domain and validating the model etc. S1, the managing element, is present in the strict guidance given to the whole process and especially to the application engineering phase.

Compared to OMT++ or XP, FAST includes a new layer, domain engineering. It represents the I2 interface in many ways. First it includes the I2.3 (knowledge sharing between a project and the base organization) while creating the application engineering environment. It could be discussed whether this is a project or a process producing this environment, but in this study it is defined as a project renewing the application engineering environment.

Knowledge sharing between the projects (I2.1) means e.g. the communication of the required changes from an application engineering project to the domain engineering project. Interface I2.2 (knowledge sharing from previous to future projects) could be defined to include sharing the application engineering environment from the domain engineering project team to the application engineering project teams. It also includes indirect knowledge sharing between application engineering projects teams through updating the application engineering environment first by the domain engineering project based on the input from application engineering projects.

Interface I1, knowledge sharing in a project team, means in this case actually two types of teams, the domain engineering team and the application engineering team. The method describes well the steps required for the actions in value adding software engineering (S2) in project team, for both the domain engineering and for the application engineering. Knowledge sharing is strongly based on artifacts.

Interface I3, knowledge sharing in customer supplier relationship, is in this case partially replaced with gaining understanding of the market and the financial feasibility of the planned product line. During an application project, the customer requirements are collected and compared to the product line environment.

In FAST I4, unofficial knowledge sharing seems to be quite similar to OMT++. The importance of documentation and clear responsibility areas is emphasized and no direct communication is really supported. Even between the people doing domain engineering and application engineering the interface consists of artifacts diminishing the need of direct communication. Thus the potential to initiate strong positive unofficial knowledge sharing seems to be low.

Project life cycle dimension (L) can, with this method, be looked at from different perspectives. When studying the domain engineering and application engineering projects as one entity, the L1 element (project establishment) could be defined to mean the whole domain engineering part required to produce the application engineering environment for the application engineering project. Then L2, project realization, would be the implementation of the application engineering projects. L3, project closure, could then be the closing of the product family. The method does not refer much to this part.

Weiss and Lai (1999, p. 154) describe that the application engineering process and the associated production facility are a result of knowledge capturing (perspective making). Knowledge, known only to a few people, is incorporated into a set of tools that can be widely distributed and used. The application engineering process captures and documents much of domain specific knowledge.

### 4.2.4 KSF Profiles of Software Development Methods

The KSF profiles of the selected methods are introduced in Figure 21 and Figure 22. The I4 (unofficial knowledge sharing) and the L (project life cycle) dimension are introduced separately without mapping with S dimension. For I4 the mapping with the S dimension is irrelevant and for L dimension it is left out for simplicity. These profiles are made based on the analysis with the KSF. Even though the + sign is used in the figures as a mark of potential for encouraging

knowledge sharing it is not a commitment of something being good (or bad). It is simply a sign of existing potential.

In OMT++ and FAST, the main way of knowledge sharing seems to be artifact (documents etc.) based, and in XP oral communication based. The KSF profile of XP (Figure 21) differs from the others with really strong knowledge sharing and communication in the customer interface (e.g. on-site-customer). On the other hand, the interface I2, knowledge sharing in the organization, is practically non-existent. OMT++ has some minor actions related to that area, but FAST has some of its strengths especially in this area and in the utilization of systematic reuse. The I2 element is crucial for organizational learning, thus the lack of this element in a method should be compensated with some additional actions in an organization.

OMT++ concentrates more on project implementation and knowledge sharing in the project team. Of course, communication with the customer is also present, but no special focus is given to it. The S1 element (managing software engineering) is normally supported with project management activities taken in addition to the phases of this method. Noticing separate project management activities also would make the profile look slightly different from S1 element perspective.

I4, the unofficial knowledge sharing (Figure 22) cannot be studied based on the method descriptions, but XP could be thought to give more place for direct communication, and thus the emergence of unofficial knowledge sharing. On the contrary, OMT++ and FAST seem more to separate people from each other through emphasizing the importance of explicitly defined artifacts and procedures.

The KSF profiles of OMT++ and XP look rather similar according to the knowledge sharing interfaces I1-I3 and the software engineering activity (S) dimension (Figure 22). In L dimension (project life cycle, Figure 22) OMT++ does not include the project closure (L3) phase. Based on this dimension, OMT++ and FAST have a similar profile even though on the other dimensions there are big differences.

A significant difference not visible in the KSF profiles is the type of knowledge sharing mainly used. In OMT++ and in FAST it is very artifact based when in XP it is mostly based on direct oral communication. This must be noticed in addition to the KSF profiles.

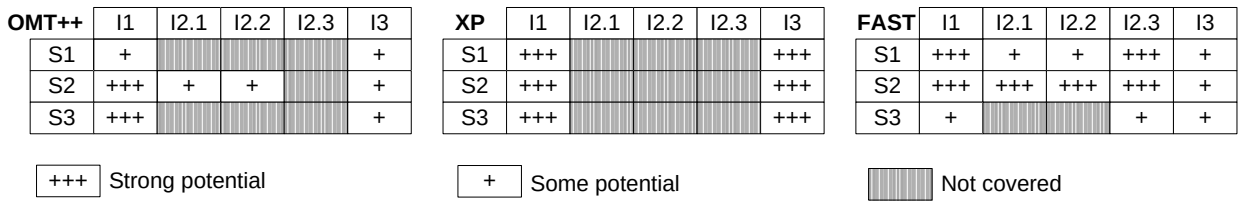



Figure 21. KSF Profiles Including I (Knowledge Sharing Interfaces) and S (Software Engineering Activity-Type) Dimensions.

| | I4 |
|---|---|
| OMT ++ | |
| XP | + |
| FAST | |

| L Dim. | L1 | L2 | L3 |
|---|---|---|---|
| OMT ++ | +++ | +++ | |
| XP | +++ | +++ | +++ |
| FAST | +++ | +++ | |

+++ Strong potential
\+ Some potential
Not covered

Figure 22. I4 and L Dimension of the KSF Profiles.

Software engineering methods are not directly interchangeable. In order to have better knowledge sharing in an organization (I2) it is not reasonable to try replacing the OMT++ or XP method with the FAST method. Instead, it is important to understand the strengths and weaknesses of the method used and to find reasonable ways to compensate for the weaknesses.

## 4.3 Summarizing

### 4.3.1 Overview

In this study the introduced Knowledge Sharing Framework (KSF) is utilized in order to study manifestation of knowledge sharing in software engineering projects in practice and also to discover the potential of selected software development methods that encourage knowledge sharing. This resulted in many findings, especially, from projects Alpha and Beta.

Some findings also came from the project survey, but clearly using a web-based questionnaire was not the most efficient way of getting a detailed-enough understanding of projects in general. Some lessons were also learned from the questionnaire design. The next time, the questionnaire is used it will need to be partially redesigned. In addition to a general web based questionnaire, short interviews could help to give a better understanding of the general situation.

The amount and quality of findings from projects Alpha and Beta, however, show the efficiency of the KSF as a tool for understanding the knowledge sharing situation in a project. Through the KSF profiles the general knowledge sharing situation can be made very visible. A combination of the KSF profiles of several projects can bring an understanding of common weaknesses and strengths in projects in an organization. Weaknesses discovered are potential targets for improvements and found strengths could be shared in the organization to repeat successes.

After starting to study the knowledge sharing situation in existing projects, the KSF was also utilized to study the potential of selected software development methods to encourage knowledge sharing. This is very important to create understanding of the knowledge sharing strengths and weaknesses of methods and to think how the potential weaknesses should be compensated when applying the methods in practice. KSF proved to be a potential tool for these kinds of evaluations also. The study showed that knowledge sharing in an organization (I2) is rarely well covered and will need some actions in addition to the methods in use.

Based on the three dimensions of KSF (knowledge sharing interfaces, software engineering activity types and project lifecycle phases), it is possible to improve the knowledge sharing

practices in software engineering. Those dimensions bring three very important perspectives of software engineering work into focus. The three-dimensional KSF proved to be efficient in identifying weaknesses and strengths in projects and processes. By adding more dimensions some new perspectives could be covered, but the three dimensions seem to be sufficient at least to get an initial understanding of the organization's knowledge sharing characteristics. Based on the results obtained with the KSF, there is always the possibility to continue with other analyses if required.

### 4.3.2 KSF and Knowledge

One potential weakness of the KSF is that it does not provide much guidance given the kind of knowledge and knowledge sharing that is really expected. As shown later in this thesis, however, the software engineering activity types (S1-S3) include some idea of the kind of knowledge involved. The main reason for not including knowledge / knowledge sharing type as one dimension is that it would have focused thinking only to certain kinds of knowledge / knowledge sharing. The reverse side is that some important kinds of knowledge / knowledge sharing might remain unnoticed.

When analyzing the KSF compared to the definition of knowledge sharing used in this study (process of perspective making and perspective taking, Section 2.1.2), the knowledge sharing interface dimension defines the interface between the communities of knowing, the interface where boundary objects are required in order to make the process of perspective making and perspective taking possible. The software engineering activity types focus our interest into three different types of software engineering activities, and actually to three different areas of knowledge sharing. The project life cycle then brings the temporal nature of projects into focus requiring knowledge sharing especially inside the organization.

With this kind of a framework, it is easier to identify existing or missing explicit knowledge sharing than tacit knowledge sharing but also the lack of tacit knowledge sharing can be made clear with the KSF. This would take place, for example, by noticing that some important knowledge exists in the organization, but, for some reason, it was not available for a project.

One way of identifying possible existing or missing knowledge sharing is to identify existing or missing communication. Communication is a sign of possible existing relationship which is one of the conditions required to have knowledge connections as defined by von Krogh (1994). To identify communication, the knowledge sharing interfaces are in key role defining the borders over which communication flows are expected to take place. KSF does not give specific support for communication analysis but is rather a complementary tool with respect to existing communication analysis methods like e.g. Social Network Analysis (Wasserman and Faust 1994).

It should be noted that when analyzing the content of communication the target of analysis is information, not knowledge. The borderline between information and knowledge is not always clear and information is often required to create knowledge in the head of the human receiver. Information is raw material for creating knowledge. So, it is reasonable to analyze also

information flows with KSF and evaluate their potential of creating knowledge in the head of the receiver.

# Part III Supporting

*In the previous part, better insight was gained to understand what knowledge sharing means or should mean in software engineering. In this part the focus is transferred from understanding to supporting knowledge sharing. First, the supporting tool to be used is selected. Next, the concepts of Knowledge Sharing Pattern and Knowledge Sharing Pattern Language are introduced, together with an account of their derivation process. Finally, the usage of knowledge sharing patterns is discussed.*

# 5 PRESENTING KNOWLEDGE SHARING PRACTICES

This chapter discusses how to present knowledge sharing practices, and introduces the knowledge sharing pattern concept as an answer for that discussion. The knowledge sharing patterns are descriptions of knowledge sharing situations targeted to support knowledge sharing in software engineering work.

In this chapter, first, the goals of knowledge sharing are defined. Next, in Section 5.2 the reasoning is given for selecting patterns as the way of describing the practices. Section 5.3 introduces the knowledge sharing pattern format and Section 5.4 introduces an example of a knowledge sharing pattern. Finally, the connection between patterns and knowledge sharing is discussed at general level.

## 5.1 Goals of Knowledge Sharing

KSF does not directly refer to certain types of knowledge shared. Indirectly, these target knowledge types are embedded into the software engineering activity types of the KSF. In this study term *target knowledge type* means the main type of knowledge involved in a knowledge sharing act. For a summary of target knowledge types, see Table 5.

S1 (managing software engineering dimension) requires knowledge sharing of work status knowledge. On the one hand it is important to know the situation compared to project plans and on the other hand the project team members should know the project plan and targets to implement the project successfully. So, in this study project plans are assumed to be part the work status knowledge.

Work status type of knowledge is very situation dependent knowledge and might go out of date fast. When that kind of knowledge is in the form of a project plan it might last a bit longer but will also need updates during a project based on growing knowledge of the requirements for the project results. Even though work status knowledge is temporary by nature, this is important type of knowledge in projects. For example, a project plan is visible document summarizing much cumulative project management related knowledge applied to a specific project.

S2 (value adding software engineering) requires knowledge sharing of requirements and work results. The requirements must be understood here as a wider concept referring to the domain knowledge in forms useful for a project. Requirements and work results knowledge is clearly value-adding knowledge describing the required results from different perspectives finally as a software code. Work results include also intermediate results like design documentation.

S3 (verifying software engineering) uses as raw material e.g. the work results, and produces findings from reviews and testing. Based on those, lessons learned (LL) type of knowledge is created.

Table 5. Deriving the Target Knowledge Types Based on the Software Engineering Activity Types.

| Software Engineering Activity Type | Target Knowledge Types | |
|---|---|---|
| **S1 Managing software engineering** | Work status knowledge (WS) | |
| **S2 Value adding software engineering** | Requirements (Reqs)<br>Work results (WR) | Work guidance (WG)<br>Competence (Comp) |
| **S3 Verifying software engineering** | Lessons learned (LL) | |

Work guidance is involved in all S elements. Work can be done without separate work guidance, but in order to assure the required quality, often some kinds of process descriptions are required to be followed in all of the S elements. One of those, S3 (verifying software engineering), when strongly simplified, means that some result is compared to its requirements. When verifying a work result, e.g. a design document, it is compared to the requirements for the project, but also to the work guidance giving the basic requirements for the software development work in the organization.

The last target knowledge type is competency type of knowledge. It is required especially in the element S2 (value adding software engineering) but like work guidance, it is involved in all of the elements. Software engineering work is based on people and their competences. Without proper competences available, improvement of the other knowledge management activities or sharing of other target knowledge types is irrelevant. Tacit knowledge in addition to explicit knowledge is included in competency types of knowledge as well as in other target knowledge types.

To support knowledge sharing the targets for the support must be defined. Knowledge sharing goals in software engineering are defined in this study in the form of a Knowledge Sharing Goal Tree (Table 6). It is built based on the target knowledge types (TK Type in Table 6) and the knowledge interfaces. First, a general knowledge sharing goal is developed based on the target knowledge type. Then, the target is divided into the knowledge sharing goals of different knowledge sharing interfaces.

As an example, see the requirements target knowledge type in Table 6. The general goal is *to understand what results can be expected in a project*. In the interface between the customer and the supplier (I3) this means: *to have a mutual understanding of the expected results of the project including the requirements for the project*. Inside a project team (I1) the target is *to know and manage the requirements among the project team members*. Inside the organization (I2) the target is to speed up delivery by systematically utilizing previously defined requirements.

It must be remembered that requirements type of knowledge is based on domain knowledge and it normally requires several iterations to be well captured for the needs of a project. The mutual understanding or requirements between the customer and the supplier personnel may need to be re-created several times during a software development project based on gained experiences including lessons learned. Also customer's needs may change during a project affecting the requirements for the project.

Table 6. The Knowledge Sharing Goal Tree for Software Engineering.

| TK Type | General goal | Interface | Knowledge Sharing Goal |
|---|---|---|---|
| **WS** | To understand the situation in projects compared to what is expected. | I1 | To follow the progress of a project. |
| | | I2 | To understand the total situation of all projects in an organization unit. |
| | | I3 | To allow the customer to know the situation in a project and to share information affecting the project in both organizations. |
| **Reqs** | To understand what results can be expected in a project. | I1 | To know and manage the requirements among the project team members. |
| | | I2 | To speed up delivery by systematically utilizing previously defined requirements. |
| | | I3 | To have a mutual understanding of the expected results of a project including the requirements for the project. |
| **WR** | To share the intermediate and final work results efficiently. | I1 | To assure efficient team work through sharing work results in a project team reasonable enough way. |
| | | I2 | To utilize previously produced work results systematically. |
| | | I3 | To share work results between the customer and the supplier. |
| **WG** | To share a common way of working to make it possible to work together and to improve the performance through improving the common way of working. | I1 | To share a common way of working in a project team. |
| | | I2 | To share, follow and improve organizational standard processes. |
| | | I3 | To have a high level of customer satisfaction through utilizing and continuously improving the common way of working in the organization. |
| **LL** | To share experiences. | I1 | To assure learning from project experiences. |
| | | I2 | To systematically share lessons learned. |
| | | I3 | To learn from the customer supplier relationship. |
| **Comp** | To have required competences available to implement the expected results. | I1 | To assure the availability of required competences in/for a project team. |
| | | I2 | To assure systematic competence support for projects and decision making situations in projects. |
| | | I3 | To define what competences are required to serve the customer in the best possible way. |

In this thesis the knowledge sharing goals are used as a guide for the kinds of knowledge sharing situations that need to be supported. Those are also used for ensuring adequate coverage of the resulting pattern language.

## 5.2 Selecting the Presentation

### 5.2.1 Needs for Support

The next step is to create support for relevant knowledge sharing in software engineering. One way to do this is to guide people to know when knowledge sharing is required and how it could be implemented. This means that an individual needs to understand when knowledge sharing is needed, and what to do in that case.

To know when knowledge sharing is required, some way of describing such a situation is needed. It can be a description of a situation in general or a description of a problem. Then this situation description would need to be followed by some kind of a knowledge sharing practice description, telling what to do in such a situation. Previously, this kind of guidance has been presented in the forms of best practice descriptions, rules of thumb descriptions and patterns. All of these have different ways of providing the description. Patterns have already been covered at Section 2.3. Rules of thumb seem normally to be pretty short statements and just covering the main idea of the solution (e.g. in Jørgensen 2007), or being a bit longer ones with do/do not do type of statements (e.g. in Peine 2005). The shorter version of a rule of thumb is too simple for the needs in this study and the longer one is still just a description of do/do not do, without giving any support, for example, in creating motivation for following the rule of thumb.

Best practices are not far from patterns if looking how those are described. As an example, McCarthy and McCarthy (2006) have described software development best practices. Their best practice #1 (pp. 11-20) is called *Establish a shared vision*. It consists of seven paragraphs of general free-form explanation and seven pages long description of one example with a summary at the end. The free-form paragraphs give mostly motivation for having a shared vision in a development team. The example gives more insight into how to create it. For the purpose of this study, this type of description is not very easy and fast to understand and to utilize in practice. It is not easy to understand quickly when to apply it and it is not giving abstract enough but still usable general solution for the situation. A more structured presentation form is required, clearly separating the different issues involved in the application of the guidance.

When given in a more structured way, a best practice description actually comes close to a pattern, presented e.g. in the Coplien format discussed in Section 2.3.2. For the purposes of this study, a well-structured pattern seems to be a good candidate for presenting knowledge sharing guidance, providing a uniform template that can be tuned to address the relevant issues.

### 5.2.2 Applicability of Pattern Format

In addition to the KSF, a knowledge sharing domain model (Figure 23) is made in order to understand what entities are related to a single knowledge sharing situation, and to show the applicability of a pattern format to describe knowledge sharing solutions. Knowledge sharing domain model is needed in this study because existing knowledge sharing models seem to concentrate more on knowledge sharing in general (e.g. Dixon 2000: leveraging common

knowledge) in an organization. In this study, it is important to understand knowledge sharing situations and what kinds of objects are involved.

The knowledge sharing domain model describes the domain of knowledge sharing. UML class diagram notation (OMG 2007) is used to present the model. According to Pender (2003, p. 108) class diagrams can be used to model nearly anything in a business or technical environment. As Fowler (2004, p. 52) says, class diagrams are very useful in exploring "the language of business" but then the notation must be kept very simple. Fowler's advice is followed in this study.

Knowledge sharing is, or at least should be, a natural part of our everyday life. It means that knowledge sharing cannot be studied without a connection to the real life actions. In the knowledge sharing domain model (Figure 23) the real life actions are referred with an entity *Act*. The word *act* is used in this study as it is used in acting plays, one set of actions in the chain of time. An *Act* may consist of several smaller *Acts* including normally several *Knowledge Sharing Acts*, pieces of the *Act* required to share knowledge. Orlikowski (2002) states, knowing cannot be separated from its constituting practice. As well, a *Knowledge Sharing Act* cannot be fully separated from the *Act*. This means that the *Act* and its *Environment* and context must be well known to understand the *Knowledge Sharing Act*.

The *Act* takes place in an *Environment* that defines the situation where the *Act* takes place. The *Environment* can be defined through its *Dimension*s, e.g. the KSF dimensions. One or more *Actor*s take part in an *Act*. What is not visible in Figure 23, is that in a *Knowledge Sharing Act* there are always at least two *Actors* required to have knowledge sharing initiated. The *Actors* do not need to be present at the same time nor be in the same place. For example, the *Acts* of creating and reading a document containing information that has the potential to create knowledge in the reader do not normally happen at the same time.

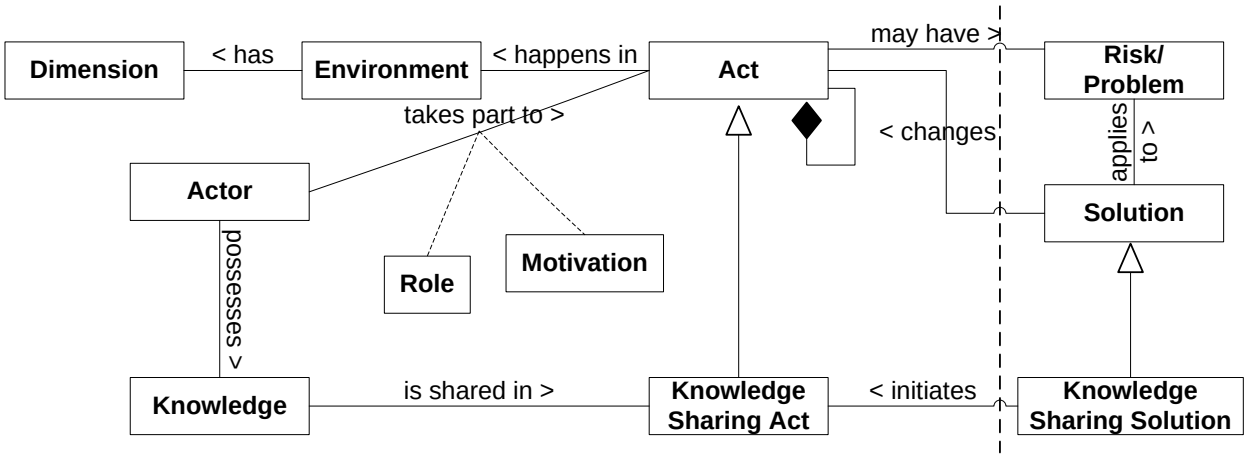



Figure 23. Knowledge Sharing Domain Model.

An *Actor* possesses some *Knowledge*. *Knowledge* then might be shared in a *Knowledge Sharing Act*, the knowledge sharing part of the *Act*. An *Actor* acts in a *Role*, and needs *Motivation* to take part in an Act. *Motivation* can vary from eagerness to participate in interesting things to earning money.

The knowledge sharing domain model would be adequate without the parts on the right after the dashed line in Figure 23. Because knowledge sharing or actually the lack of it becomes visible in the form of different kinds of problems, however, this part is added. Very often, an *Act* involves some *Risks* or *Problems*. A *Solution* to the problem or a solution to decrease the severity of risk might be known. Normally, it involves some kind of a procedure as to what to do and the motivation to use to find the solution. The solution normally includes also *Knowledge Sharing Solution* as part of that solution. The *Knowledge Sharing Solution* initiates a *Knowledge Sharing Act*.

As mentioned, one of the entities in the knowledge sharing domain model (Figure 23) is the *Risk/Problem*. Very often the lack of knowledge sharing is visible in the form of a problem or there is a risk that should be avoided. The approach in this study, to support knowledge sharing, is primarily to initiate knowledge sharing thus preventing some risks known to realize and to create problems. When a risk is high or a problem has occurred several times, it is reasonable to introduce a common solution for that problem. Patterns, by nature, include a problem and solution pair making it a very good candidate here. The problem statement of a pattern tells what kinds of problems or what kinds of risks of problems can be solved with the pattern.

The knowledge sharing domain model has many similarities to pattern structures. When also noticing this added problem/risk based approach, the similarity is very clear. E.g. the Coplien (1996) format maps to the knowledge sharing domain model very well. Figure 24 shows the main classes of a pattern description described the same way as the knowledge sharing domain model. The Context in Figure 24 refers to both to the context and the resulting context of, for example, Coplien (1996). The class Context remains even though applying the Solution changes the Context (from initial to resulting context).

The part on the right, Problem and Solution can be described the same way as in the knowledge sharing domain model (Figure 23). The Context includes nearly all other parts in the knowledge sharing domain model, especially the *Act*, the *Environment*, the *Dimension* and the *Actor* in *Role* and possessing *Knowledge*. The Forces (in Figure 24) could be thought to be a possible source to create *Motivation* mentioned in the knowledge sharing domain model.

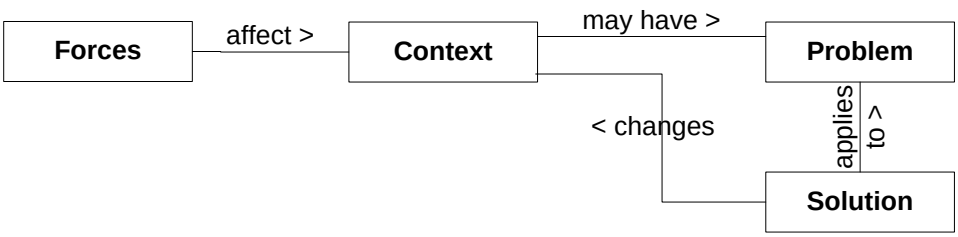



Figure 24. Main Parts of a Pattern Description.

A very challenging part of initiating knowledge sharing is how to motivate people to share knowledge. It requires an organizational culture that favors knowledge sharing. The pattern format alone does not fully solve the motivation creation challenge, but it does give some support. Compared to e.g. just a simple procedure description, how to do something, the pattern format includes also a description of forces that affect in the situation. Understanding the forces might support the commitment to and approval of the introduced pattern. The context and resulting context, the environment before and after applying the solution, show how the solution would affect the forces. Patterns link different levels of thinking (Winn and Calder 2002) and the forces have a major role in this. In addition to noticing that pattern format can bring some kind of support for the motivational part, the pattern format seems to be a good selection as a tool for improving knowledge sharing in software engineering.

### 5.2.3 Patterns or Just Using Pattern Format?

Similarly like Paasivaara (2005, pp. 196-197) discusses whether her communication practices are practices or patterns, in this study we could question whether the resulting knowledge sharing patterns are patterns in the most disciplined meaning of the term pattern (e.g. by Coplien 1996 and Coplien and Harrison 2005), or just practices described using pattern format.

To call something a pattern it should be identified in more than one environment. That is not the case for all of the resulting patterns. The justification principles used here are introduced later in this thesis (Section 10.2). These principles ensure certain level of practical evidence of the benefits of the patterns, but they are less demanding when it comes to observed instances of the patterns. Also, not all patterns work totally in isolation but they need other patterns to complete the action. Although all patterns are at a general level independent entities, in some cases we have found to be important to give more support for single actions in patterns in form of another pattern. In such cases also the supporting pattern has independent role, but it supports building the bigger entity of the main pattern. These kind of supporting pattern relationships are marked with dashed arrows in Figure 30, Figure 31, Figure 32, Figure 33, Figure 34, and Figure 35.

Even when knowing all these weaknesses we decided to call the resulting as patterns. The main reason for this is that for people working in the sector of software engineering, the term pattern creates the right context and understanding of the purpose of the tool. On the other hand, the resulting patterns are still in the early stages of their evolution, requiring further collaborative elaboration – they could be called prototype patterns. Those are, however, adequate from the perspective of this study, allowing the recording of findings of this work in a systematic way.

## 5.3 Knowledge Sharing Pattern (KSP) Format

### 5.3.1 Format Overview

The format used for describing patterns in this study is based on Coplien's (1996) format. It is adapted to suit the requirements of describing knowledge sharing patterns. Other formats could

have been selected as the basis as well, but this one is chosen because of its clarity and simplicity. Coplien format very nicely guides us to notice the most important things related to patterns.

The used format is introduced in Table 7. Some of those items originate directly from Coplien's (1996) format, such as *Pattern Name*, *Problem*, *Context*, *Forces*, *Solution* and *Resulting Context*. The context is renamed the *Initial Context*. Some new items are added, like: *Dimensions*, *Knowledge Flow*, *Roles*, *Instances* and *Process Connection*. The item rationale from the Coplien's format is left out here for reasons of simplicity.

Item *Dimensions* is used to define the relationships between a single pattern and the pattern language as described in the knowledge sharing domain model. Item *Knowledge Flow* introduces the main knowledge flow: source, destination and (if applicable) the key actor (the role written with bold letters). In addition to this main knowledge flow, a pattern normally has other knowledge flows. Item *Knowledge Flow* defines the *Knowledge Sharing Act* introduced in the knowledge sharing domain model. Roles participating in the implementation of the pattern are introduced better in the item *Roles* referring to the *Role* in the knowledge sharing domain model. Item *Instances* are instances of *Acts* having the K*nowledge Sharing Act* highlighted. Those are examples of implementations of knowledge sharing patterns in practice.

As item *Dimensions* is for connections to the pattern language, so the item *Process Connections* is for connections to the processes of the company. The *Process Connection* then links instances of *Acts* with the process environment of an organization. This is important because the knowledge sharing patterns are aimed to support the use of the processes and to add new features to those processes.

The item *Related Patterns* refers to some existing patterns by other authors having similarities with the knowledge sharing pattern in question and/or supporting the pattern. In the resulting patterns described in Appendix B, some examples are given. The knowledge sharing patterns have been developed first and after that some existing sets of organizational patterns were studied and linked to relevant knowledge sharing patterns. There is no detailed account of the relationships, since such analysis is beyond the purposes of this thesis and not essential for a user.

The knowledge sharing patterns have also an identifying code of type *KSP<nn>*, where KSP is a constant and <nn> refers to consecutive numbering. The numbering is based on the order in which the knowledge sharing patterns have been created.

The items listed in Table 7 are the items of the actual knowledge sharing pattern description. In addition to these, knowledge sharing pattern descriptions may include other required topics such as justification and references. These other topics give more background information or examples.

Table 7. The Knowledge Sharing Pattern Format.

| Item | Explanation |
|---|---|
| **Pattern Name** | The name of the pattern. |
| **Dimensions** | Dimensions from the Knowledge Sharing Pattern Language defining the relation of the pattern with the pattern language.<br>Knowledge sharing interface<br>Target knowledge type<br>Project Lifecycle phases, unless all |
| **Knowledge Flow** | The main knowledge flow: source, destination and main actor (if applicable). |
| **Problem** | A brief description of the problem. |
| **Initial Context** | The situation to which the pattern solution applies. |
| **Roles** | The roles implementing the pattern. |
| **Forces** | Forces that affect the situation. |
| **Solution** | The required instruction to solve the problem in the context. |
| **Resulting Context** | The situation/context which will result from performing the pattern solution. |
| **Instances** | Utilization of this pattern, examples of use. Also some potential pitfalls in the use of this pattern. |
| **Process Connection** | The connections of the pattern to the organization's processes in use. |
| **Related Patterns** | Introducing related patterns from other authors (optional). |

### 5.3.2 Describing the Solution

The solution part of a knowledge sharing pattern starts with a picture introducing the solutions steps. Following the picture is a more detailed description of the steps. UML activity diagrams (OMG 2007) are used to introduce the steps.

According to Fowler (2004, p. 117) activity diagrams can be used to describe the business process and the work flow which is the case here. In addition, the swimlanes nicely describe the roles and make knowledge sharing very visible in activity diagrams. Therefore, activity diagrams are a standard notation that is used in this study to describe procedures participated in by several stakeholders.

Knowledge sharing happens when an arrow or an action crosses a swimlane border. As an example, see Figure 25. It is the activity diagram from the knowledge sharing pattern *Created Bones* (KSP06). First, as is shown in Figure 25, it is not possible to separate the knowledge sharing act totally from the act. So, to introduce the knowledge sharing act, a part of the act needs to be introduced. Also, it is not always clear which actions could be counted as preparation for knowledge sharing (part of perspective making) and which are just parts of the act but not part of the knowledge sharing act. Because of this, the solutions are built based on the act, but concentrate on describing the knowledge sharing parts of it with special attention.

As indicated, knowledge sharing takes place, when an arrow or an action crosses a swimlane border at an activity diagram. In Figure 25 there are two kinds of examples of this. First there is

an arrow crossing a swimlane border from action *Ask for Requirements* to action *Tell Requirements*. This kind of an arrow means communication, where the responsibility for the next action is clearly transferred from one role to another role. Another example is the action that crosses the swimlane borders. For example, the action *Gather Requirements* is implemented by all the roles that take part in this action. They work (and communicate) together to have the action implemented.

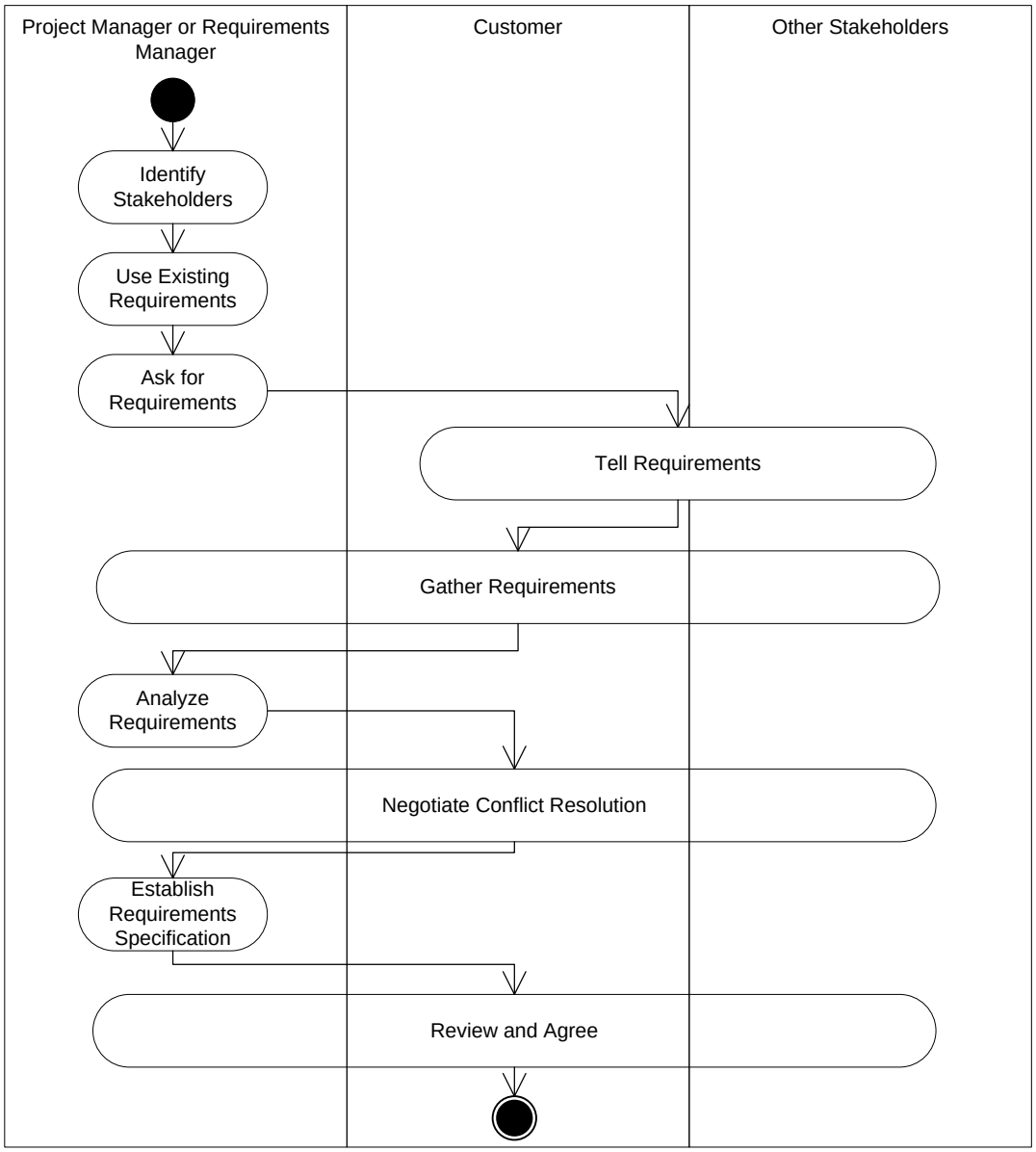



Figure 25. An Activity Diagram Example from the Knowledge Sharing Pattern *Discovered Bones* (KSP06).

Artifacts (e.g. input and output from actions) are not introduced in the knowledge sharing pattern activity diagrams. The main reason for this is that together with the swimlanes the artifacts would make the diagrams complex. E.g. if an artifact would be utilized by several roles, the artifact should be drawn over the swimlane borders. In addition to making the picture complex, strict definition of artifacts could hinder the generic nature of a diagram and leave out agile solutions for an activity. The defined activity diagrams include some activities that clearly will result in an artifact, even though, the artifact is not separately presented in the picture. The action Establish Requirement Specification in Figure 25 is one example of this.

## 5.4 An Example of a Knowledge Sharing Pattern

Knowledge sharing pattern *Shared Understanding* (KSP05) is introduced next as an example. For other knowledge sharing pattern descriptions, see Appendix B. Knowledge sharing pattern *Shared Understanding* (KSP05) is selected as an example because it is very important among the patterns. For example, several other patterns refer to this pattern or require this as an initial context. First a screenshot is given from the web page of the pattern (Figure 26) and then the actual pattern description.

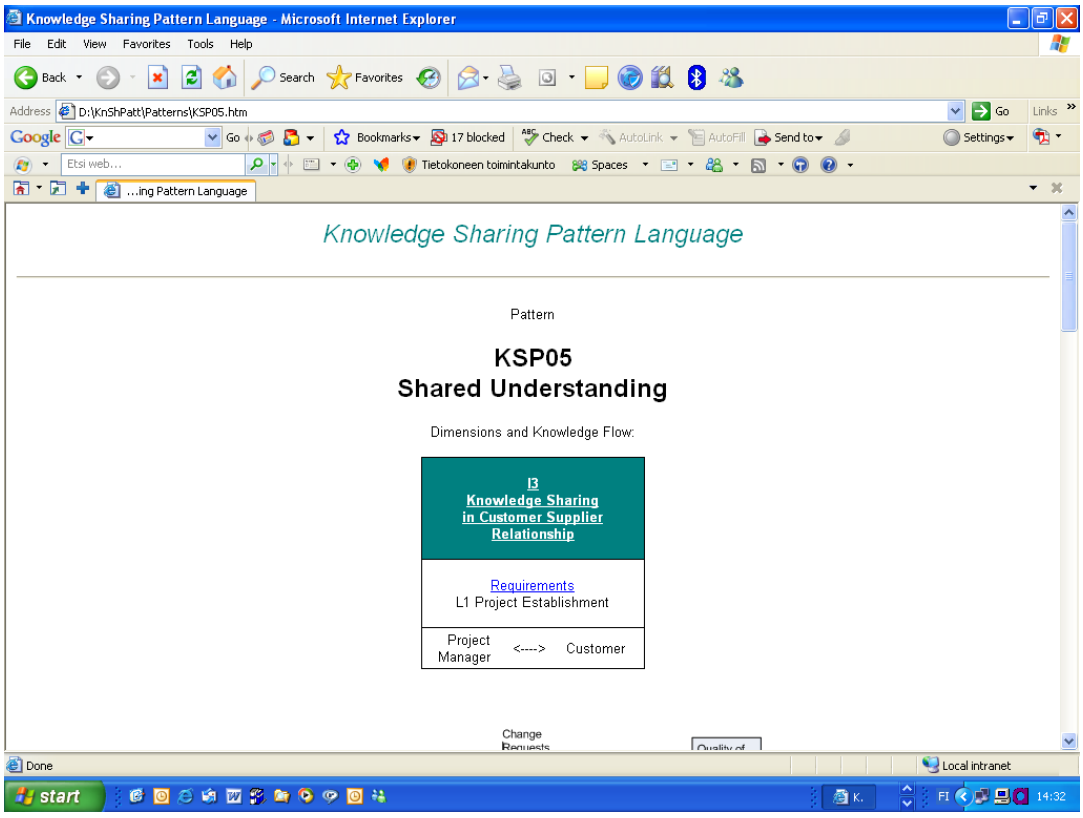



Figure 26. A screenshot from pattern *Shared Understanding* (KSP05).

**Dimensions**

- **I3 Knowledge Sharing in Customer Supplier Relationship**
- **Requirements**
- **L1 Project Establishment**

**Knowledge Flow**

| Project Manager | <----> | Customer |
|---|---|---|

**Problem** The expected results of a project are not yet clear enough for all parties, especially between the customer and the supplier.

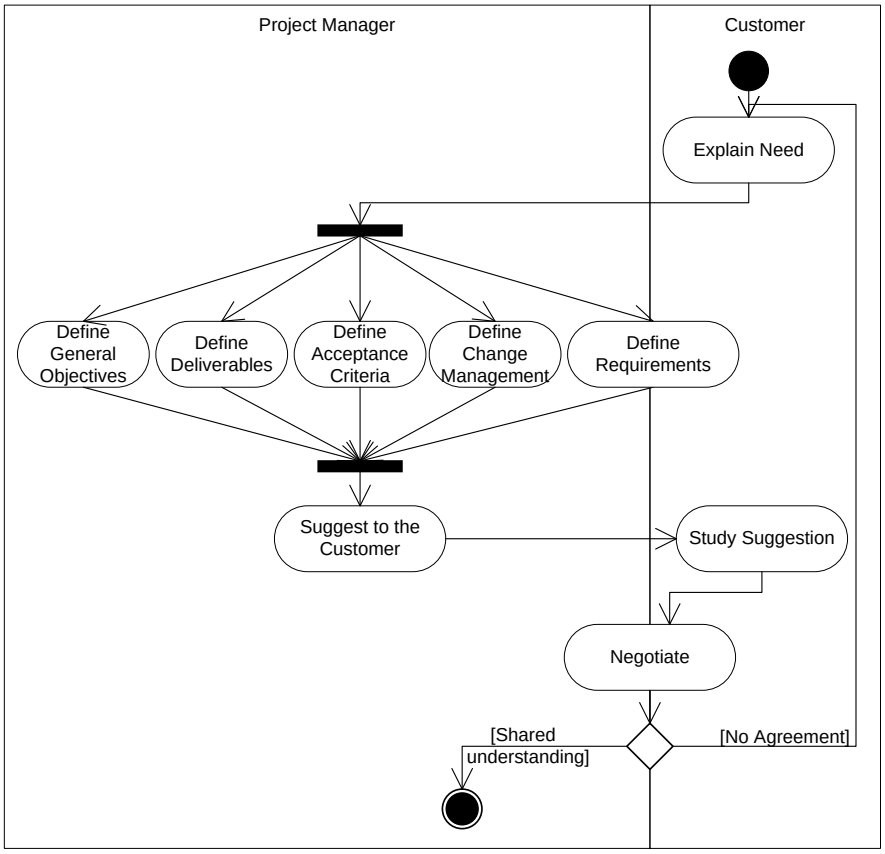
**Initial Context** A new software engineering project is planned or the expected results of a current project are not clear enough to the customer and/or to the supplier.

**Roles** *A Project Manager* or alike, perhaps also a sales representative together with *a Customer Representative* define the aimed results of the project.

**Forces** In the establishment phase the project must be first defined in the form of required results. The results must be based on the customer's needs and the supplier must have clear understanding of what the expected results are. Detail planning of a project is not possible before the general objectives are defined. A more detailed requirements definition is possible only after first having the understanding of the general objectives.

Normally, in practice, it is very difficult to define when a software engineering project can be closed. Not having well defined acceptance criteria might result in a customer and a supplier having totally different expectations. Finally, change normally cannot be avoided during a software engineering project.

**Solution**



To define and agree on the required results:

1. Customer *explains* the *need* (in practice, this continues parallel to Step 2).
2. Supplier defines suggestion of :
    - *General objectives* for the project
    - *Deliverables*: what exact deliverables are required (software, documents, services,...)
    - *Acceptance criteria*: how the results are to be accepted and using what criteria? This could include e.g. a definition of the required quality level of the result or the amount and type of defects allowed such as "no blockers".
    - *Change management*: how changes to the defined requirements, results etc. will be initiated, processed and approved.
    - *Requirements*: what functions etc. are required (See Discovered Bones, KSP06). Remember that requirements capture is in most cases iterative activity and requires involvement of several stakeholders. In the beginning of a project, however, some kind of early understanding is needed about the requirements or at least about the ways of capturing those.
3. Supplier makes a *Suggestion to the customer* and the customer *Studies the Suggestion*.
4. *Negotiate* to establish shared understanding of results for a project. If the shared understanding is achieved, then this pattern ends. If not, then the customer has to continue explaining the need. Shared understanding can mean an agreement of results, or an agreement that solution together (customer and supplier) is impossible.

Note that this is a simplified process defining the required results. Very often this is done parallel with the commercial discussions. Also the requirements definition can be done at this phase or later e.g. as a separate pre-study phase.

**Resulting Context** Shared understanding created between the customer and the supplier about the aimed results of a project. Procedures and approval rights approved for processing changes. Clear acceptance criteria defined making possible an objective judgment of the project readiness.

**Instances** The pattern is expected to be used always when project results are to be defined. Normally, this is in the beginning of a project or during the sales phase before a project.

One potential pitfall is that the acceptance criteria are not detail enough to really be the basis when judging the project readiness.

**Process Connection** Project establishment and sales.

In the example above, as in several other knowledge sharing patterns, the Knowledge Flow includes actor Customer and in some other patterns an actor Organization. Officially these refer to a Customer Representative and an Organization Representative respectively. These have been shortened for practical reasons.

## 5.5 Patterns and Knowledge Sharing

Patterns have several kinds of connections to knowledge sharing and different types of knowledge. Those are discussed in this section. First, it is important to understand the connections in general, and second to understand how knowledge sharing patterns bring one additional knowledge sharing layer to patterns.

Creating and utilizing a pattern is one way of knowledge sharing. When a pattern is a knowledge sharing pattern, knowledge sharing takes place at two levels. First, sharing a pattern, or in other words sharing a description of how to do something, and at the second level, the initiation of knowledge sharing through implementing the knowledge sharing pattern solution. Thus, a knowledge sharing pattern could be defined as an introduction to a meta level of knowledge sharing.

### 5.5.1 Knowledge Sharing and Pattern Creation & Utilization

Vlissides (1998, p. 8) describes a pattern to be "a vehicle for capturing and conveying expertise, whatever the field". Thus, already by nature, a pattern is a tool for sharing knowledge. Patterns help to solve problems that could be introduced with how to questions. According to Dixon (2000, p. 11) this is one sign of common "know how" knowledge. Thus patterns could be defined as representative of common knowledge in the meaning Dixon (2000, p. 11) has used this term.

Figure 27 introduces the creation and utilization of patterns at a very general level. Many different reasons may initiate a need to define a pattern. Here, it is assumed that it started by noticing that several project teams confront a very similar problem <1> (see the numbers in Figure 27). Possible solutions <2>, real life best practices, are searched and identified. Based on those, a new pattern description <3> is created. Finally the pattern description is utilized by projects <4> having the problem (or risk of it) and the context defined in the pattern description.

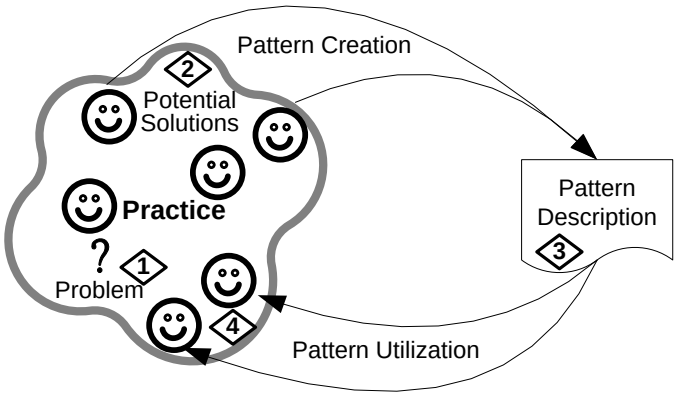



Figure 27. Creation and Utilization of Patterns.

May and Taylor (2003) have looked at the same matter from slightly different angles. Their starting point is that if knowledge cannot be handled directly in databases, the focus needs to be in improving the processes where information is converted to knowledge. They have explored the use of patterns as a powerful mechanism for doing that.

Based on Nonaka and Takeuchi's (1995) interplay between tacit and explicit knowledge, May and Taylor (2003) have introduced the use of patterns supporting it (Figure 28). Pattern mining supports knowledge conversion between tacit and explicit knowledge. Similarly actions to support pattern writing (shepherding and writer's workshops) support this conversion. Selecting, using and reflecting on pattern use support the knowledge conversion from explicit to tacit knowledge. Tacit to tacit knowledge conversion could take place e.g. in mentoring relationships. The Explicit to explicit conversion takes place through combining patterns, and through mixing those with other knowledge representations.

May and Taylor's (2003) introduction of patterns in interplay between tacit and explicit knowledge is actually pretty much the same as in Figure 27. If replacing the pattern description (in Figure 27) with the explicit knowledge, and replacing the practice with the tacit knowledge, these explanations would still work. As a result we can notice that these explanations are well in line with each other.

Table 8 introduces the connections of pattern creation and utilization to selected knowledge sharing theories. Boland and Tenkasi's (1995) action called perspective making can be mapped directly to the creation of a pattern. In the language of Nonaka (1994) this is the knowledge transformation process called externalization, the transfer from mostly tacit knowledge (how people work) to explicit knowledge (pattern description), or explicit-to-explicit (combination) conversion. The creation of common knowledge and the first two phases of the leveraging common knowledge, as defined by Dixon (2000), could be thought to be the process of creating a new pattern based on a real life practice.

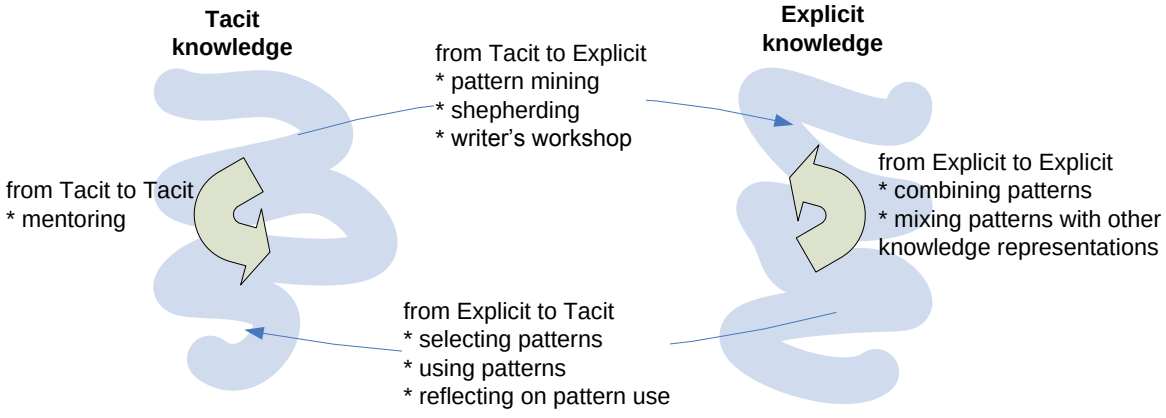



Figure 28. Patterns and Knowledge Interplay (reproduced from May and Taylor 2003, p. 95)

Table 8. Knowledge Sharing Activities and pattern Creation & Utilization.

| Source | Creation of a pattern <2> | Pattern description <3> | Utilization of a pattern <4> |
|---|---|---|---|
| **Boland and Tenkasi, 1995** | Perspective making | Boundary object | Perspective taking |
| **Nonaka et al. 2006** | Externalization<br>Combination | (result of previous knowledge conversion and source for the next) | Internalization<br>Combination |
| **Dixon, 2000** | Creating common knowledge and two first phases of Leveraging common knowledge (selecting knowledge transfer system and translating knowledge) | Gained and structured common knowledge | Rest of the phases of leveraging common knowledge |

Because real life is very rich with features, a big part of knowledge cannot be captured to the pattern description. A pattern description is always an abstraction and, already by the nature of patterns, a pattern must be generic enough to produce reproducible solutions to defined problems. When using Boland and Tenkasi's (1995) terminology, the pattern description (or the main terms of it) could be defined to be a boundary object making it possible for different communities of knowing to understand each other and to share the knowledge. From the perspective of Dixon (2000), the pattern description is explicit documentation describing the gained common knowledge.

The utilization of a pattern by e.g. a project team then conforms to Boland and Tenkasi's (1995) perspective taking and Dixon's leverage of common knowledge. As highlighted by Dixon (2000, p. 21), the receiving team might even improve on the pattern solution resulting in an improved pattern. In Nonaka's (1994) terminology, knowledge conversion takes place from explicit to tacit knowledge (internalization) or from explicit to explicit knowledge (combination).

In addition to be knowledge sharing between people, a pattern can also create an 'Aha!' affect. It means situations where a person understands that the same has been in his latent knowledge, but the pattern has reminded him about that and helps him to utilize that earlier latent knowledge in practice.

Connections between patterns and knowledge sharing could also be looked at from the perspective of the type of knowledge found in different phases of knowledge sharing. Mapping of some knowledge types is introduced in Table 9. For the numbers in diamonds at the table, see Figure 27.

Based on von Krogh et al. (1994) the main target in knowledge sharing is sharing of private knowledge to be organizational knowledge in practice. Existing organizational knowledge needs to be shared with new members of the organization. In addition, through pattern descriptions, new private knowledge is also created. The texts in parentheses show the potential of other knowledge types while the main target is sharing private knowledge to be organizational knowledge.

According to von Krogh et al. (1994) everything known is known by somebody. This means that a pattern description would not practically contain knowledge. According to Dixon (2000, p. 13) the pattern description would be information instead of knowledge. Then by processing information the receiver could create knowledge.

Table 9: Knowledge Types from von Krogh et al. (1994), Blackler (1995), and Cook and Brown (1999) .

| Source | ② Knowledge in practice before perspective making | ③ Knowledge in pattern description | ④ Knowledge in practice after perspective taking |
|---|---|---|---|
| **Von Krogh et al. 1994** | Private<br>(Organizational) | None | Organizational<br>(Private) |
| **Blackler 1995** | Embrained<br>Embodied<br>Embedded<br>(Encultured)<br>(Encoded) | (Encoded) | Embedded<br>Encultured<br>Encoded<br>(Embrained)<br>(Embodied) |
| **Cook and Brown 1999** | Skills: individual-tacit<br>Genres: group-tacit<br>(Concepts: individual-explicit)<br>(Stories: group-explicit) | (Group / explicit) | Stories: Group-explicit<br>Genres: Group-tacit<br>(Concepts: Individual-explicit)<br>(Skills: Individual-tacit) |

From the perspective of Cook and Brown (1999), Nonaka's (1994) knowledge conversions, externalization and internalization, are not real knowledge conversions between different types of knowledge. They state that knowledge should not be treated as being of one kind. Instead of separate types of knowledge, they have introduced four forms of knowledge. According to their forms, the perspective making could be defined to be a transfer from all forms of knowledge, but especially from individual tacit skills and group tacit genres to explicit group stories. This means that the raw material for a pattern is the tacit skills and the way of working including also possible explicit definitions etc. The pattern itself then could be defined to be the explicit group story (or no knowledge at all). The perspective taking part could be defined to be the same transfer to the opposite direction.

A good pattern, when utilized in practice, should affect all forms of existing knowledge possessed by people and an organization (= all people together). For the individual-explicit form of knowledge (concepts) a pattern could initiate new rules and concepts. Individual-tacit knowledge (skills) could be affected, for example, through piloting new ways of doing things and achieving new skills during that process. The pattern description itself would be an addition to group-explicit knowledge (stories). The pattern name could then become as new shared meaning for the group and it might later on be used when referring to certain types of actions. While pattern description is very general the pattern name could get deeper meanings inside the group and then the pattern name is attached with certain types of group-tacit (genres) knowledge.

When using the knowledge types of Blackler (1995) (Table 9), we could say that one goal with patterns is to make visible embrained and embodied (individual) knowledge and make it possible to use as group knowledge. Also embedded knowledge is one important source for organizational patterns, not to forget the other types of knowledge either.

The pattern descriptions themselves could be defined to be no knowledge at all or an example of encoded knowledge. Utilization of an organizational pattern can result in changes in an organization's processes and in that way result in changes in embedded knowledge. The patterns quite often start functioning as a vocabulary of shared meanings and emerge as encultured knowledge. Utilization of a pattern may affect also other types of knowledge.

### 5.5.2 Knowledge Sharing Pattern – a Meta Level of Knowledge Sharing

The target for knowledge sharing patterns is to initiate relevant knowledge sharing as part of normal software engineering work. Knowledge sharing patterns include solutions about how to share knowledge in certain situations. Knowledge sharing patterns are here employed as vehicles to share knowledge about the ways of sharing knowledge. It is actually a meta level of knowledge sharing.

Sharing of explicit knowledge is relatively easy, if an explicit presentation, as a boundary object, is understandable as such to those receiving it. In addition it should not be counted as 'noise' by the receiver and being of interest for the receiver. As a knowledge sharing pattern solution this includes the following steps:

1. Person possessing knowledge documents the knowledge as information.
2. Information (e.g. a document) is made available for a receiver needing it.
3. Knowledge of the receiver is created through processing the received information.

An important part of knowledge is tacit knowing. It is knowledge that cannot be easily described as documented/verbally communicated information. Tacit knowledge poses new challenges. It is possible, however, to initiate the sharing of tacit knowledge with knowledge sharing patterns. The sharing is just not as direct as it would be with explicit knowledge. One possible way to facilitate the sharing of tacit knowledge is to guide the person in need of knowledge to observe an identified expert or through other ways to make it possible to observe experts in their work. In these cases, the knowledge sharing pattern solution is actually a guidance to where and how to observe. One example of this is the resulting knowledge sharing pattern *Named Experts* (KSP02, introduced in Appendix B).

Normally, knowledge sharing situations are very rich in practice, including sharing both explicit and tacit knowledge. Thus, these are not separately introduced in the knowledge sharing patterns. For example, sharing requirements type of knowledge includes e.g. documented explicit requirements but also much undocumented tacit knowledge that is relevant to what the documented requirements mean in practice.

# 6 KNOWLEDGE SHARING PATTERN LANGUAGE

This chapter introduces the Knowledge Sharing Pattern Language utilizing the knowledge sharing pattern concept which was introduced in the previous chapter. Section 6.1 discusses the basics of Knowledge Sharing Pattern Language forming a pattern language. Next, Section 6.2 introduces the structure of the pattern language. The resulting pattern catalog is briefly introduced in Section 6.3. Finally, the two main views of the pattern language are introduced in Section 6.4. Those are: the target knowledge stream view and the knowledge sharing interface view. All patterns are linked to these two views with the exception of I4 (unofficial knowledge sharing) patterns because of their unofficial nature and also the target of knowledge sharing.

## 6.1 Introduction

A pattern language is here defined based on the definition from Schuler (2008, p. x): a pattern language is an organized collection of patterns that together express a broad coherent response to a number of related problems. The coherent whole the pattern language creates is *a system of efficient knowledge sharing in a software engineering organization*. Knowledge Sharing Pattern Language supports this especially in small and medium size organizations. Like nearly all pattern languages, this pattern language does not ensure that all possible relevant actions will take place. However, the language is expected to point out the key hot spots of knowledge sharing in a software development process and provide guidance on how to manage them.

All patterns in Knowledge Sharing Pattern Language follow the knowledge sharing pattern format introduced in Section 5.3. The syntax, the order and the way of using single patterns, varies slightly based on the different usages introduced in Chapter 8. The structure of the pattern language introduced in the next section is a sort of map of the pattern language. It describes different pairs of knowledge sharing interfaces and target knowledge types that need to be covered to ensure efficient knowledge sharing in a software development organization. The order of implementing those can be changed based on the needs in the organization.

A pattern language is never complete, but it evolves and improves over time. This is also the case with the resulting Knowledge Sharing Pattern Language. Considerable effort has been devoted to get the individual patterns and the pattern language in the present form, but their further improvement is an ongoing process.

## 6.2 Structuring the Pattern Language

The basis for the structure of Knowledge Sharing Pattern Language (Figure 29) is the KSF and its elements. The knowledge sharing interfaces are the same as in the KSF. In addition, target knowledge types are defined based on the software engineering elements in the KSF (see Section 5.1). The project lifecycle element is not visible at the main structure level. Instead, it is described at a single pattern level, when a pattern is specific for some of the lifecycle elements.

As a result, the Knowledge Sharing Pattern Language is a matrix of knowledge sharing interfaces and target knowledge types. This provides the basic structure for the Knowledge Sharing Pattern Language. These knowledge sharing interfaces and target knowledge types form their own clusters of knowledge sharing patterns. The matrix structure of the Knowledge Sharing Pattern Language is given in Figure 26. The parent pattern KSP00 supports the establishment of this structure and binds the dimensions together.

KSF includes three dimensions: knowledge sharing interfaces, software engineering elements and project life cycle elements. These are all visible in the knowledge sharing pattern format. Every pattern is linked to one knowledge sharing interface element (I). The only exception is the knowledge sharing pattern *Work Guidance* (KSP22, introduced in Appendix B) being linked to interfaces I1-I3. The knowledge sharing interface element is introduced in the dimension table in the beginning of a pattern description. The interface stakeholder roles are also visible in the knowledge flow, in role definition, and as separate swimlanes in the activity diagrams.

The links to software engineering elements (S) in the KSF are visible in the knowledge sharing pattern format through the target knowledge types (Section 5.1). A target knowledge type of a knowledge sharing pattern is introduced in the dimension table in the beginning of a knowledge sharing pattern description. Knowledge sharing interfaces and target knowledge types are also the main structuring levels of the Knowledge Sharing Pattern Language. That way those also link single patterns to the whole pattern language. The same dimension table also introduces the project life cycle phase to which the pattern applies to. The linking between patterns in general and the knowledge sharing domain model was defined earlier in Section 5.2.2.

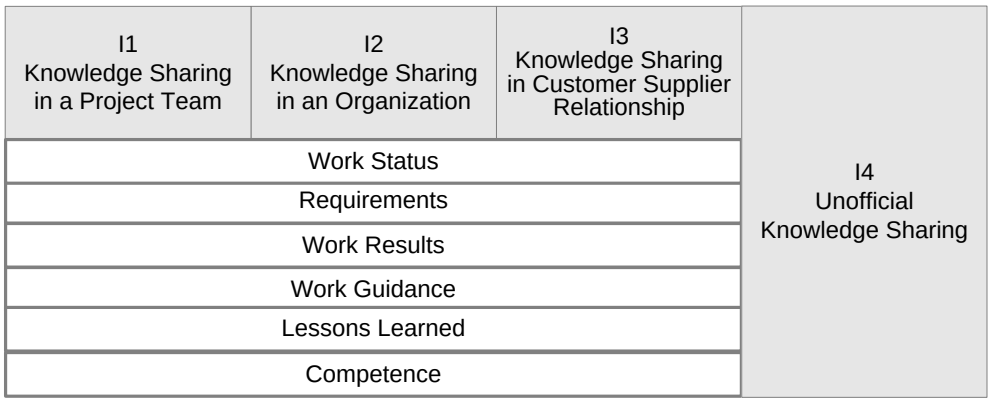



Figure 29. The Structure of the Knowledge Sharing Pattern Language.

A pattern language is often implicitly structured according to the context specifications which may create relationships between the patterns: if one pattern provides a context required by another pattern, the latter pattern is potentially applicable after the former. Although initial and resulting context specifications are used in the format of knowledge sharing patterns, these specifications do not consciously aim at an implicit structuring of the language, but they are given purely as advice for the pattern user.

It could be discussed if the target knowledge types or knowledge sharing interfaces could be used to form separate pattern languages, but in this study, we have decided to introduce all of these patterns as one Knowledge Sharing Pattern Language. The matrix of these two dimensions gives a map of the pattern language, making it possible for an actor to identify the relevant patterns on the basis of the knowledge sharing interfaces and target knowledge types involved in the situation at hand.

## 6.3 Pattern Catalog

The Knowledge Sharing Pattern Language includes, at this moment, 28 actual patterns and one parent pattern which is also a homepage for the Knowledge Sharing Pattern Language web. A website allows easy linking between patterns and also from the organization's process materials.

The knowledge sharing patterns and their problem statements are introduced in Table 10. Those are introduced in the order they were developed. For detailed knowledge sharing pattern descriptions, see Appendix B.

Table 10. The Catalog of Knowledge Sharing Patterns.

| Id | Pattern Name | Pattern Problem |
|---|---|---|
| **KSP00 (parent)** | Improved Knowledge Sharing | An organization wanting to make work more efficient through improving knowledge sharing. |
| **KSP01** | Project Support | A project manager is often very alone in a decision making situation and has to make a decision without adequate knowledge. The organization might have required knowledge elsewhere, but did not know how to make it available for projects in need. |
| **KSP02** | Named Experts | An organization not knowing how to make available the knowledge and experiences of some of its members to benefit many projects and the whole organization. |
| **KSP03** | Assigned Experts | A project team is missing certain competence areas that are required in a project. |
| **KSP04** | Trust or Check | A project manager is unsure about how to follow a project team member's progress in a project. |
| **KSP05** | Shared Understanding | The aimed results of a project are not yet clear enough for all parties, especially between the customer and the supplier. |
| **KSP06** | Discovered Bones | Customer and the supplier not having common understanding of the requirements for the project. |
| **KSP07** | Reference Requirements | No knowledge about what requirements definitions might already exist and how to find those as potential reusables for projects. |

| Id | Pattern Name | Pattern Problem |
|---|---|---|
| **KSP08** | Created Skeleton | Project team members having only very general knowledge about what the results of a project should be. |
| **KSP09** | Schedule Baseline | A project manager cannot start proper project progress follow-up, because she/he does not have defined tasks nor a schedule for comparison with the current status. |
| **KSP10** | Known Status of Projects | Not knowing adequately enough the general situation of projects in an organization and thus no possibilities to manage the situation with several projects. |
| **KSP11** | Informed Customer | Customer and supplier not having adequate common understanding about the project status and/or not communicating adequately about changes in both organizations affecting the project through interdependencies. |
| **KSP12** | Managed Versions | Difficulties in sharing the (intermediate) work results in the project team. |
| **KSP13** | Quickly Made | Time available is very limited to have the work results implemented and having the defined level of quality. |
| **KSP14** | Release | The delivery of (intermediate) work results between the customer and the supplier has not been organized or is not working well. |
| **KSP15** | Discovered Lessons | In a project many things are learned but those are not systematically collected and understood. |
| **KSP16** | Established Experience Base | Lessons learned are collected in projects but the organization does not have any systematic way to store and share those to support the work in projects. |
| **KSP17** | Satisfied Customer | The project manager does not know how the customer perceives the project and its results. |
| **KSP18** | Improved Competences | A project manager does not know how well the competences of the project team match the competence requirements of the project. |
| **KSP19** | Assured Resources | An organization not being sure if it has available adequate resources for a new project. |
| **KSP20** | Initiated Communication | Communication structures of an organization need to be strengthened. |
| **KSP21** | My Network | Not strong enough personal communication network. |
| **KSP22** | Work Guidance | An organization with a need to establish or improve the guidance of work in the organization in order to allow for more efficient team work. |
| **KSP23** | Not Wasted | Requirements for a new project must be defined, but time available is very limited. |
| **KSP24** | Flexible Skeleton | During a project, requirements change and the project team members do not know what changes must be implemented and how those affect the project. |
| **KSP25** | Followed Progress | A project manager not knowing the project situation and progress status well enough. |
| **KSP26** | Reuse Approach | No knowledge about what work results might already exist and how to find those as potential reusables for projects. |
| **KSP27** | Contributed Experience Base | Lessons learned are collected in projects but not shared at the organizational level. The same problems are occurring again and again in different projects. |
| **KSP28** | Utilized Experience Base | Projects are not utilizing the lessons learned that are collected in the organization. The same problems are occurring again and again in different projects. |

## 6.4 Two Views to the Knowledge Sharing Pattern Language

### 6.4.1 Target Knowledge Streams

The target knowledge streams introduce acts related to certain types of target knowledge. The origin of the target knowledge types is introduced in Section 5.1. The term *stream* is used here to highlight the stream type of continuous process and knowledge flow related to a target knowledge type.

Every target knowledge stream is introduced separately in this section. Part of the introduction is the target-knowledge specific part of the Knowledge Sharing Goal Tree, including the patterns related to it.

#### 6.4.1.1 Work Status

Work status knowledge refers to all knowledge supporting the understanding of the progress of a project, including plans. Table 11 introduces the work status part of the Knowledge Sharing Goal Tree completed with the patterns supporting the knowledge sharing of work status knowledge. It includes five patterns.

At the knowledge sharing interface I1 the goal is to follow the progress of one project. Three knowledge sharing patterns are created to implement this target. First, some kind of *Schedule Baseline* (KSP09) is required to know what is planned and how the plan is to be implemented. Then the *Progress* must be *Followed* (KSP25), so that it can be compared to the Schedule Baseline. The follow-up of single project team members might need some support. That support is given in *Trust or Check* (KSP04) pattern.

Table 11. Work Status Stream and Knowledge Sharing Patterns.

| TK Type | General goal | Interface | Knowledge Sharing Goal | Knowledge Sharing Patterns |
|---|---|---|---|---|
| WS | To understand the situation in projects compared to what is expected. | I1 | To follow the progress of a project. | KSP09 Schedule Baseline<br>KSP25 Followed Progress<br>KSP04 Trust or Check |
| | | I2 | To understand the total situation of all projects in an organization unit. | KSP10 Known Status of Projects |
| | | I3 | To allow the customer to know the situation in a project and to share information affecting the project in both organizations. | KSP11 Informed Customer |

At the interface I2 the target is to understand the total situation of all projects in an organizational unit. The knowledge sharing pattern *Known Status of Projects* (KSP10) is introduced for this purpose. At the interface I3 the pattern *Informed Customer* (KSP11) had the target of having the customer knowing the project situation and both parties sharing information that might affect the project through interdependencies.

Figure 30 introduces the pattern references to each other. Patterns with gray are from outside the work status stream. Inside this group of patterns the pattern *Schedule Baseline* (KSP09) is central having links to nearly all other patterns in this stream. It is a very important pattern in practice, because it supports the utilization of other patterns

#### 6.4.1.2 Requirements

Requirements knowledge refers to the knowledge about what is required as a result of a project. The main part of Requirements Knowledge, therefore, is the requirements for a project. In general, it also includes the shared understanding with the customer about the expected results of the project. A metaphor named *bone* is used for single requirements. The requirements (bones) together then form a *skeleton*. In addition to bones and the skeleton, there is also the need for joints in the skeleton. The knowledge of the expected results in the project, in general, acts as the joints.

Table 12 introduces the requirements part of the Knowledge Sharing Goal Tree completed with the patterns supporting the knowledge sharing of requirements knowledge. It includes six patterns.

First, it is important to assure that there is shared understanding with the customer about the expected results of a project. This takes place in interface I3 and has two knowledge sharing patterns: *Shared Understanding* (KSP05) for general level understanding and *Discovered Bones* (KSP06) for requirement level understanding.

When defining the requirements, the organization (I2) should assure efficiency through the possibility of utilizing earlier-defined requirements. Two knowledge sharing patterns are introduced to support this: *Reference Requirements* (KSP07) for initiating collecting reusable requirement sets and *Not Wasted* (KSP23) for the utilization of those in a single project.

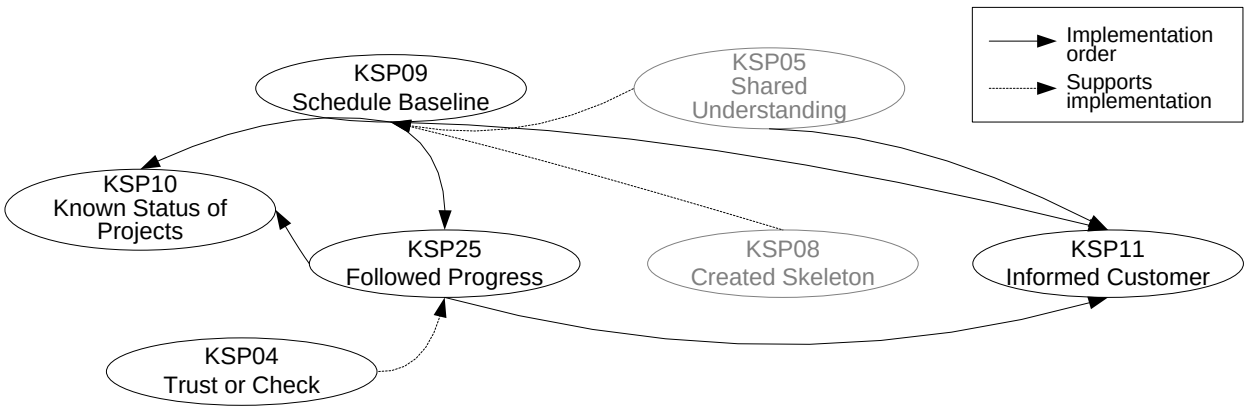



Figure 30. Pattern References from the Work Status Stream Patterns.

Table 12. Requirements Stream and Knowledge Sharing Patterns.

| TK Type | General goal | Interface | Knowledge Sharing Goal | Knowledge Sharing Patterns |
|---|---|---|---|---|
| Reqs | To understand what results can be expected in a project. | I1 | To know and manage the requirements among the project team members. | KSP08 Created Skeleton<br>KSP24 Flexible Skeleton |
| | | I2 | To speed up delivery by systematically utilizing previously defined requirements. | KSP07 Reference Requirements<br>KSP23 Not Wasted |
| | | I3 | To have a mutual understanding of the expected results of a project including the requirements for the project. | KSP05 Shared Understanding<br>KSP06 Discovered Bones |

Also, it is important to assure that the project team (I1) knows the requirements and that the requirements are managed. Two patterns support these expectations: *Created Skeleton* (KSP08) for gaining the knowledge and *Flexible Skeleton* (KSP24) for managing.

Figure 31 introduces the pattern references to each other. In practice, and also by observing most of the arrows in Figure 31, the pattern *Shared Understanding* (KSP05) is the most important pattern on which the others are all somehow based. Without proper shared understanding with a customer, a project cannot be implemented successfully. This is especially important, because, in most of the cases, the main metric for success is customer satisfaction.

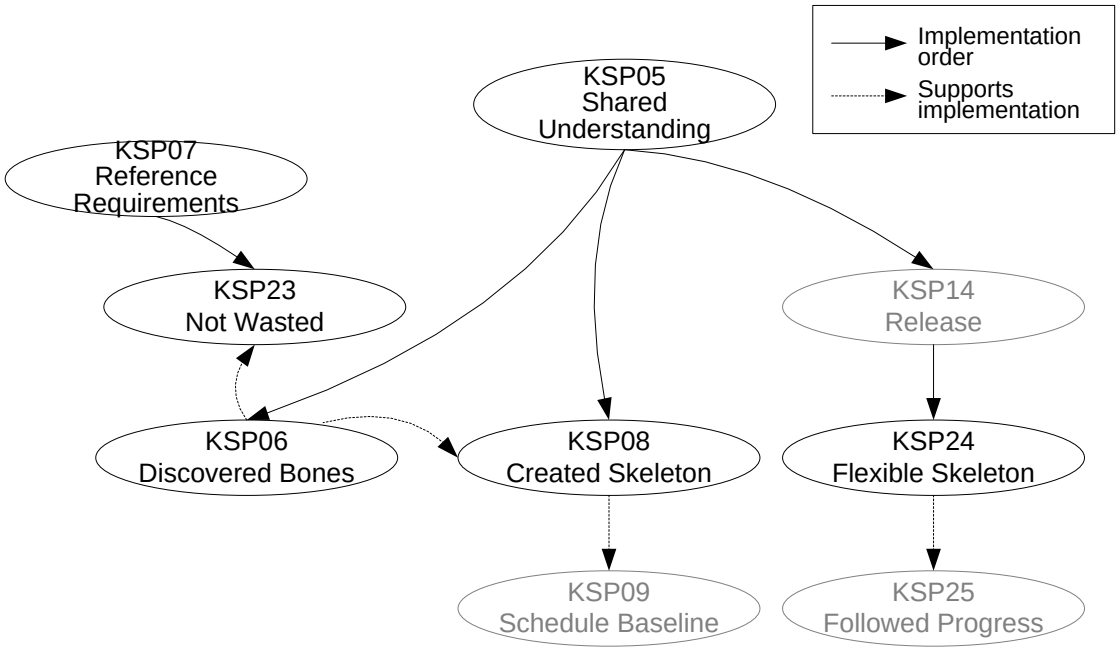



Figure 31. Pattern References from the Requirements Stream Patterns.

#### 6.4.1.3 Work Results

The work results stream refers to all kinds of value adding artifacts that are results from the software engineering work. This does not cover only the final results but also the intermediate results created and used during a project. Table 13 introduces the work status part of the Knowledge Sharing Goal Tree completed with the patterns supporting the knowledge sharing of work results knowledge. It includes four patterns.

In order to share the work results efficiently, we must first define what we expect those results to be and how we want to share them in a project team (I1). Pattern *Managed Versions* (KSP12) serves this purpose.

When producing the work results, more speed could be gained through reusing results that were produced earlier. That step needs organization (I2) level support. Patterns *Reuse Approach* (KSP26) and *Quickly Made* (KSP13) are introduced for this purpose. Finally, the work results need to be shared with the customer. Pattern *Release* (KSP14) serves that purpose.

Figure 32 introduces the pattern references to each other. In this stream the patterns are more independent of each other than in earlier streams. The pattern *Shared Understanding* (KSP05) is important here also even though it is from another stream. The pattern *Managed Versions* (KSP12) could be thought to be one key pattern in this stream because it has connections also to interfaces I2 and I3. It belongs to interface I1.

Table 13. Work Results Stream and Knowledge Sharing Patterns.

| TK Type | General goal | Interface | Knowledge Sharing Goal | Knowledge Sharing Patterns |
|---|---|---|---|---|
| WR | To share the intermediate and final work results efficiently. | I1 | To assure efficient team work through sharing work results in a project team reasonable enough way. | KSP12 Managed Versions |
| | | I2 | To utilize previously produced work results systematically. | KSP26 Reuse Approach<br>KSP13 Quickly Made |
| | | I3 | To share work results between the customer and the supplier. | KSP14 Release |

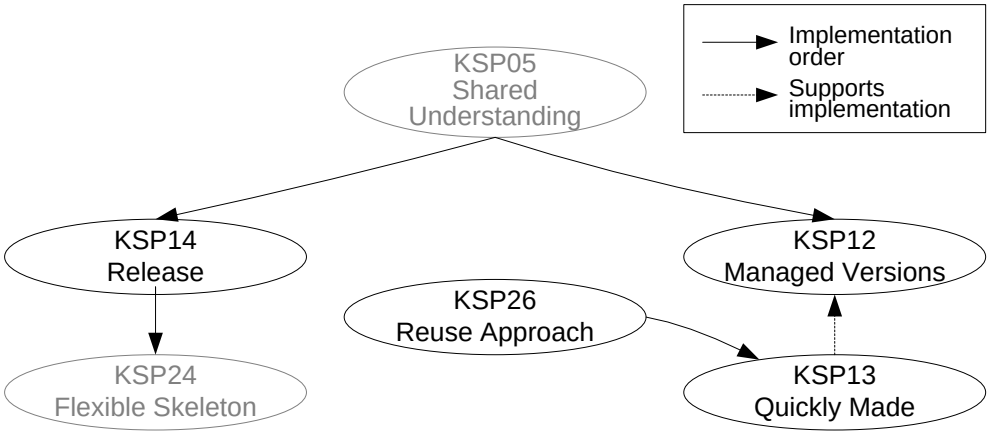



Figure 32. Pattern References from the Work Result Stream Patterns.

#### 6.4.1.4 Work Guidance

The work guidance stream refers to all knowledge guiding the work. Table 14 introduces the work guidance part of the Knowledge Sharing Goal Tree complete with the pattern supporting that kind of knowledge sharing.

The main target for knowledge sharing in this stream is to share a common way of working to make it possible to work together and to improve performance through improving the common way of working. The general goal is then divided into sub-targets for knowledge sharing in different knowledge sharing interfaces. After studying many different approaches to this stream, the final result was to have only one pattern, *Work Guidance* (KSP022) to serve this purpose. It seems to work in that purpose nicely, because all three interfaces need the same target and process-driven basis.

Table 14. Work Guidance Stream and Knowledge Sharing Pattern.

| TK Type | General goal | Interface | Knowledge Sharing Goal | Knowledge Sharing Patterns |
|---|---|---|---|---|
| WG | To share a common way of working to make it possible to work together and to improve the performance through improving the common way of working. | I1 | To share a common way of working in a project team. | KSP22 Work Guidance |
| | | I2 | To share, follow and improve organizational standard processes. | |
| | | I3 | To have a high level of customer satisfaction through utilizing and continuously improving the common way of working in the organization. | |

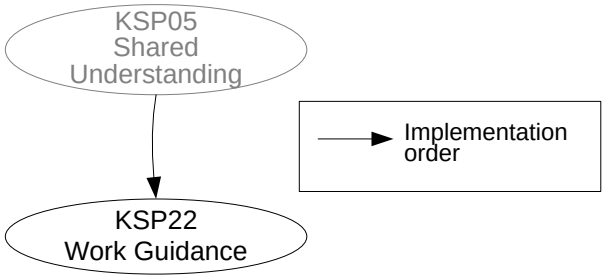



Figure 33. Pattern References from the Work Guidance Stream Pattern.

Figure 33 introduces the pattern reference from the *Work Guidance* (KSP22) pattern. As noted in earlier streams, the pattern *Shared Understanding* (KSP05) has an important role here also.

#### 6.4.1.5 Lessons Learned

Lessons learned knowledge refers to the common knowledge resulting from the experience in projects. Knowledge sharing patterns are sort of lessons learned type of knowledge. Table 15 introduces the lessons learned part of the Knowledge Sharing Goal Tree complete with the patterns supporting the knowledge sharing of lessons learned knowledge. It includes five patterns.

To share the lessons learned effectively, first those need to be discovered in a project team. Pattern *Discovered Lessons* (KSP15) serves this purpose. Another source for potential lessons learned is in the customer supplier relationship. Pattern *Satisfied Customer* (KSP17) represents this interface. Finally, the lessons learned need to be shared in the organization. For that, three patterns are introduced: *Establish Experience Base* (KSP16), *Contributed Experience Base* (KSP27) and *Utilized Experience Base* (KSP28).

Table 15. Lessons Learned Stream and Knowledge Sharing Patterns.

| TK Type | General goal | Interface | Knowledge Sharing Goal | Knowledge Sharing Patterns |
|---|---|---|---|---|
| LL | To share experiences. | I1 | To assure learning from project experiences. | KSP15 Discovered Lessons |
| | | I2 | To systematically share lessons learned. | KSP16 Established Experience Base<br>KSP27 Contributed Experience Base<br>KSP28 Utilized Experience Base |
| | | I3 | To learn from the customer supplier relationship. | KSP17 Satisfied Customer |

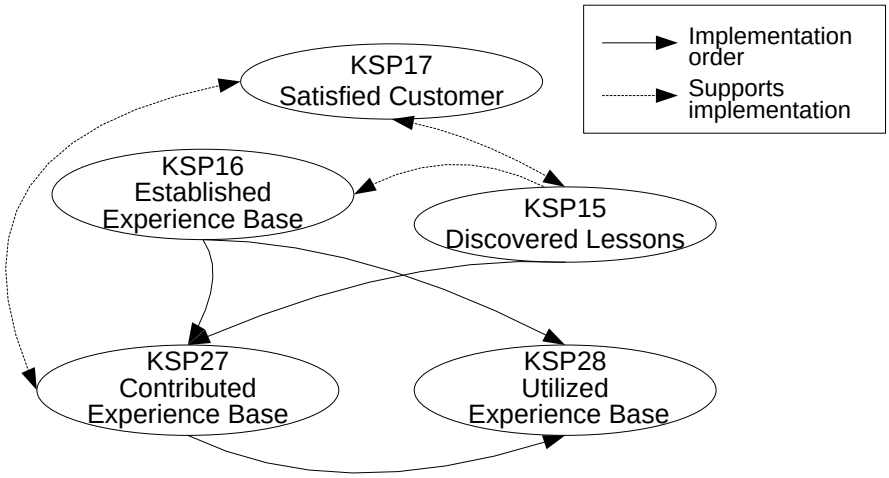



Figure 34. Pattern References from the Lessons Learned Stream Patterns.

Figure 34 introduces the pattern references to each other. The pattern Discovered Lessons (KSP15) is here a sort of a basic pattern. Without it, the lessons learned would not exist. The patterns *Established Experience Base* (KSP16), *Contributed Experience Base* (KSP27) and *Utilized Experience Base* (KSP28) have close connections and actually those are all different parts of the act of leveraging lessons learned knowledge.

#### 6.4.1.6 Competence

The competence stream refers to the competences of people in an organization or in a project. Table 16 introduces the competence part of the Knowledge Sharing Goal Tree completed with the patterns supporting the knowledge sharing of competence type of knowledge. It includes five patterns.

Table 16. Competence Stream and Knowledge Sharing Patterns.

| TK Type | General goal | Interface | Knowledge Sharing Goal | Knowledge Sharing Patterns |
|---|---|---|---|---|
| **Comp** | To have required competences available to implement the expected results. | I1 | To assure the availability of required competences in/for a project team. | KSP18 Improved Competences |
| | | I2 | To assure systematic competence support for projects and decision making situations in projects. | KSP01 Project Support<br>KSP02 Named Experts<br>KSP03 Assigned Experts |
| | | I3 | To define what competences are required to serve the customer in the best possible way. | KSP19 Assured Resources |

First, it is important to understand what resources are required and does the organization have adequate resources to serve the customer in certain planned project. Knowledge sharing pattern *Assured Resources* (KSP19) serves this purpose.

The resources of an organization should be used as efficiently as possible. Very often all resources required cannot be assigned to one project only since they may need to be available for more projects. Pattern *Project Support* (KSP01) serves the purpose of arranging organizational competence support to projects. Patterns *Named Experts* (KSP02) and *Assigned Experts* (KSP03) implement two main parts of the pattern *Project Support* (KSP01).

On site at a project it is important to identify what competence requirements can be met by the project team. For example you may discover that additional training is required. It is important to understand, as well, what other competences are required. Pattern *Improved Competences* (KSP18) serves this purpose.

Figure 35 introduces all pattern references from the competence stream patterns. The patterns outside this stream are drawn with gray. The pattern *Project Support* (KSP01) could be defined to be the main pattern for these. There are connections to all the patterns in this stream through its parts (KSP02 and KSP03). The amount of dashed arrows in Figure 35 indicates that in this stream the pattern entities could be slightly rethought when making next revision of this pattern language. Otherwise the isolation of single patterns would not be good enough.

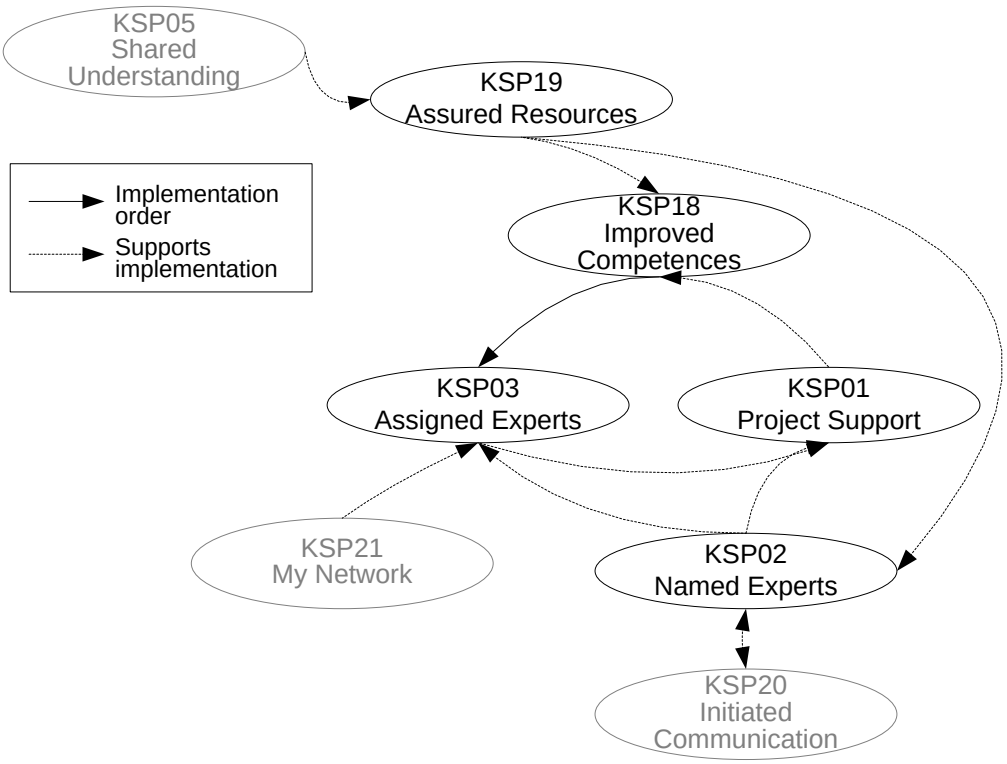



Figure 35. Pattern References from the Competence Stream Patterns.

### 6.4.2 Knowledge Sharing Interfaces

In addition to the target knowledge sharing streams, the knowledge sharing interfaces have also acted as an interface to the knowledge sharing patterns. Every pattern is linked to one knowledge sharing interface. The only exception is the pattern *Work Guidance* (KSP22) that is linked to three interfaces: I1-I3.

The interfaces I1-I3 are all internally divided into the target knowledge streams. The interface I4, Unofficial Knowledge Sharing, does not have this kind of division inside of it because of its nature. The patterns linked to I1-I3 have already been introduced so, here, those are just listed per each interface. The interface I4 has some more explanation, because the patterns related to it have not been introduced earlier.

#### 6.4.2.1 Knowledge Sharing in Project Team

The interface I1, knowledge sharing in a project team, refers to knowledge sharing between project team members. The patterns related to it are listed in Table 17.

#### 6.4.2.2 Knowledge Sharing in an Organization

Interface I2, knowledge sharing in an organization, refers to knowledge sharing between projects or between a project and the base organization. In practice, this means knowledge sharing between a project team member and a person in another project or, for example, an accounting person or line manager in the organization. The patterns related to it are listed at Table 18.

Table 17. I1 Knowledge Sharing Patterns.

| Target Knowledge Stream | Knowledge Sharing Patterns |
|---|---|
| **Work Status** | KSP09 Schedule Baseline<br>KSP25 Followed Progress<br>KSP04 Trust or Check |
| **Requirements** | KSP08 Created Skeleton<br>KSP24 Flexible Skeleton |
| **Work Results** | KSP12 Managed Versions |
| **Work Guidance** | KSP22 Work Guidance |
| **Lessons Learned** | KSP15 Discovered Lessons |
| **Competence** | KSP18 Improved Competences |

Table 18. I2 Knowledge Sharing Patterns.

| Target Knowledge Stream | Knowledge Sharing Patterns |
|---|---|
| **Work Status** | KSP10 Known Status of Projects |
| **Requirements** | KSP07 Reference Requirements<br>KSP23 Not Wasted |
| **Work Results** | KSP26 Reuse Approach<br>KSP13 Quickly Made |
| **Work Guidance** | KSP22 Work Guidance |
| **Lessons Learned** | KSP16 Established Experience Base<br>KSP27 Contributed Experience Base<br>KSP28 Utilized Experience Base |
| **Competence** | KSP01 Project Support<br>KSP02 Named Experts<br>KSP03 Assigned Experts |

#### 6.4.2.3 Knowledge Sharing In Customer Supplier Relationship

Interface I3, knowledge sharing in customer supplier relationship, refers to knowledge sharing between the supplier (e.g. project manager) and the customer representatives. The patterns related to it are listed in Table 19.

Table 19. I3 Knowledge Sharing Patterns.

| Target Knowledge Stream | Knowledge Sharing Patterns |
|---|---|
| **Work Status** | KSP11 Informed Customer |
| **Requirements** | KSP05 Shared Understanding<br>KSP06 Discovered Bones |
| **Work Results** | KSP14 Release |
| **Work Guidance** | KSP22 Work Guidance |
| **Lessons Learned** | KSP17 Satisfied Customer |
| **Competence** | KSP19 Assured Resources |

#### 6.4.2.4 Unofficial Knowledge Sharing

Interface I4, unofficial knowledge sharing, refers to all unofficial knowledge sharing between individuals. In this study the effort is more on the other three interfaces, but a couple of examples are developed also for this interface.

Unofficial knowledge sharing is not easy or straightforward to request by the organization. It happens based on an individual's own will and motivation. Of course, one's own will and motivation is required for all knowledge sharing, but in other interfaces there can be guidance given because of the more formal nature of the knowledge sharing.

As we discussed earlier, there are also some ways to guide the unofficial knowledge sharing. One example is the information security restrictions and the guidance related to those restrictions. By implementing such restrictions, an organization tries to avoid leaking critical business information to other parties. The opposite way is trying to initiate it. At the Knowledge Sharing Pattern Language, there are two patterns to support unofficial knowledge sharing. The first one of those, *Initiated Communication* (KSP20) is targeting to initiate (especially) unofficial knowledge sharing by introducing people to each other. The aim is to give them possibilities to meet each other and to form sub-communities inside the organization and between organizations. This pattern is created from the viewpoint of an organization.

The other knowledge sharing pattern is *My Network* (KSP21). It is created from an individual's perspective. The target of this pattern is to support the creation of a personal network.

# 7 DEVELOPING KNOWLEDGE SHARING PATTERN LANGUAGE

This chapter describes how the knowledge sharing patterns and the Knowledge Sharing Pattern Language have been developed. Section 7.1 describes the search, writing and validation of patterns. This section introduces also a brief history of how the resulting patterns have been created in Sub-Section 7.1.4. The Section 7.2 summarizes the whole process of creating Knowledge Sharing Pattern Language. It also introduces the same process at a bit more general level making it available also to other domains.

## 7.1 Developing Knowledge Sharing Patterns

The process for developing single patterns (Figure 36) includes activities: pattern search, writing and validation. This is an iterative process that also continues after the introduction of the first official baseline of the Knowledge Sharing Pattern Language. For example, Coplien and Harrison (2005) had processed and utilized their organizational patterns for ten years before publishing those as a book for a wider audience. As knowledge is seen here as continuously developing, also the knowledge sharing patterns should continuously be improved, especially by persons utilizing those in practice.

The purpose of the pattern search phase is to identify potential patterns from practice and other sources. After identifying a potential pattern the pattern needs to be described and written as a pattern description. Finally the pattern needs to be validated. The search for patterns is guided by practical needs and/or by the planned structure of the pattern language. These phases are introduced in more detail in the next three sections.

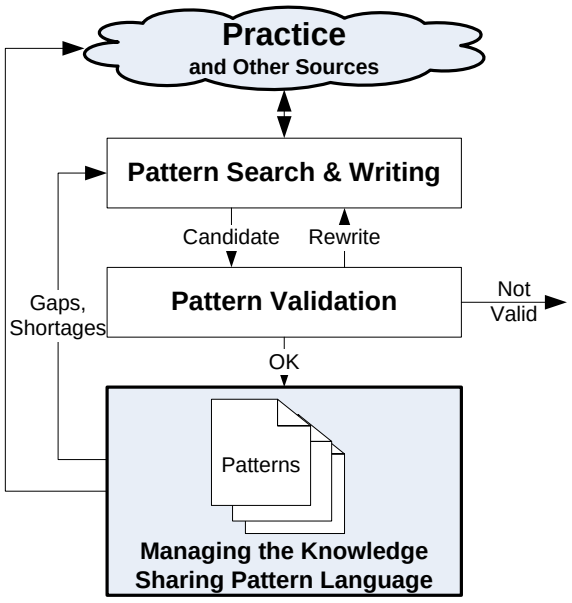



Figure 36: Developing Knowledge Sharing Patterns

### 7.1.1 Knowledge Sharing Pattern Search

A pattern cannot just be invented: it must be based on something existing. Without that kind of a link it would not be a pattern. In this study, several sources have been used to search patterns. The primary sources have been practice, methods from literature and organizational standard processes. The study of practice has resulted in the identification of several problems and best practices regarding knowledge sharing. The problems are a sign of a possible need for new patterns. The best practices may be candidates for patterns. One example of a purely practice-based approach is the pattern *Trust or Check* (KSP04, Appendix B). This pattern has originated from the need for such from projects such as Alpha and Beta, as well as the resulting search for how other project managers solve this. There were more than one possible solution so the main criteria used had to be selected among those and the pattern written based on those selections.

As patterns are abstractions, similarly methods, especially software development methods, are abstractions that can be sources for patterns. Some instructions for a method also might be steps for a pattern solution directly or after some modifications. An example of the use of these kinds of sources could be the pattern *Discovered Bones* (KSP06, Appendix B). This pattern actually covers the requirements development phase for a project. For that several practically proven methods exist and the pattern is written based on those.

Another source is the standard processes used in the company. A discussion could emerge on whether or not those standard processes are really followed in practice. In this study, however, the definition is that if the implementation of the processes is followed by audits that do not result in significant non-conformance reports, the processes are accepted to be in use. Through observing company practice additional confidence can be gained. An example is the pattern *Shared Understanding* (KSP05, Appendix B). There the steps follow the standard process of the organization being studied. This practice was found in use in many cases in the studied

organization. In at least one of those cases (project Alpha) it was discovered that this process was not being applied systematically but that the implementation of it would have helped.

In addition to these sources, existing patterns as defined by others (e.g. Coplien and Harrison 2005), could be used. Several already-defined patterns exist in literature. These, too, are potential candidates as a source for knowledge sharing patterns. In some cases they could be used almost as they are, needing just slight restructuring to suit the format used in this study. In other situations, they might need more modifications. In this version of the knowledge sharing patterns, existing patterns have not yet been utilized as a source, but some similar patterns created by other authors have been introduced in the *Related Patterns* in some of the pattern descriptions (see Appendix B).

### 7.1.2 Writing Knowledge Sharing Patterns

After finding a potential pattern based on the sources, it needs to be introduced as a pattern description, at this time called a *pattern candidate*. When producing the knowledge sharing patterns a set of pattern writing guidelines was defined and used to ensure a certain minimum quality level for the patterns. The first of those guidelines is: *A knowledge sharing pattern needs to follow the knowledge sharing pattern format*. E.g. Vlissides (1998, pp. 147-148) highlights the importance of a certain structure to be used for all patterns of the same pattern language. The format that was used is described in Section 5.3. When using this guideline for checking the resulting pattern description additional detail check items have been used to ensure that the format really is followed.

The second guideline is: *A name of a pattern needs to be noun based or otherwise reasonable, short, and good for a vocabulary*. This follows the guidance given by Coplien (1996). Pattern names are very important because those must work well as terms in daily discussions when referring to the pattern. The pattern name refers then to a piece of shared knowledge in a community of knowing.

The third guideline is: An *initial context needs to give adequate understanding of preconditions and the beginning state.* Coplien (1996) highlights this. A properly defined initial context is critical to understand if the pattern is applicable or not in the case.

The fourth guideline is: *Relevant forces are identified, documented, and solution balances the forces some reasonable way.* Forces play an important role in many of the parts of a pattern format. Those give deeper understanding (Winn and Calder 2002) to the pattern, the solution must balance the forces (Coplien and Harrison 2005), and finally the resulting context tells which forces have been solved and which have not (Coplien 1996).

The fifth guideline is: *A pattern is easy to understand*. Vlissides (1998, p. 150) says: "The quality of your patterns is determined by how well you present them." A pattern needs to be written in a clear and unpretentious style. Pattern description cannot be useful for people if they do not understand it. Also, very often, the readers need to read patterns with some other than their native language. Simple but effective language helps the understanding. Refining a pattern can continue forever. At some point, however, the patterns need to be shown to others and really put to use.

The last guideline is: *A pattern is a relevant addition to the pattern language and works also as a separate entity*. In addition to being and inspiring format to share knowledge, Alexander (1999) defines patterns having other dimensions. One of those is generativity. It means the capability of a pattern language to produce coherent wholes. When there are many patterns, a whole pattern language, the emergent behavior of complex systems can be captured (Winn and Calder 2002). Single knowledge sharing patterns are required to define single knowledge sharing situations but together those can improve knowledge sharing in software engineering.

Coplien and Harrison (2005) report that a single pattern captures locally related concerns and one pattern can be applied without concern for other patterns in the language. Patterns are always part of some whole that gives them context (Coplien and Harrison 2005, p. 6). Together the patterns of a pattern language capture the domain's key concepts and the important aspects in their relations (Winn and Calder 2002).

Similarly, several other guidelines could have been defined, but these were found to be most important ones in this study. These guidelines are very useful when checking the pattern descriptions after writing those. The way these guidelines have been applied to the pattern *Trust or Check* (KSP04) is introduced in the next list:

1. *A knowledge sharing pattern needs to follow the knowledge sharing pattern format*: Used structure follows the knowledge sharing pattern format. The following content updates were made based on this check:
   - L dimension was missing and L2 Project Realization was added.
   - The knowledge flow was checked. Attention was especially given to the direction of the arrow (from project team member to the Project Manager), because this should show the main knowledge flow, not necessary all knowledge flows in the situation.
   - No potential pitfalls were introduced, so those were added.
2. *A name of a pattern needs to be noun based or otherwise reasonable, short, and good for a vocabulary*: Name *Trust or Check* is not noun based, but it is short and describes very nicely the solution.
3. An *initial context needs to give adequate understanding of preconditions and the beginning state*: Initial context of the pattern clearly stated the situation. The pattern is meant to be used in addition to the normal basic reviews etc.
4. *Relevant forces are identified, documented and solution balances the forces some reasonable way*: The definition of forces could always be improved, but those are now defined adequately. Those describe the main forces affecting in this context from the perspective of this problem. One force, the potential negative results of surveillance was not explicitly solved and a note was added to the resulting context.
5. *A pattern is easy to understand*: The pattern is described in a very clear way and the activity diagram supports the understanding of the pattern description. Of course, it must be noticed that every user of the pattern makes his or her decision about whether or not a pattern is easy to understand. The pattern solution, forces etc. should be quite easy to understand from the pattern description.
6. *A pattern is a relevant addition to the pattern language and works also as a separate entity*: The pattern works well separately (= clear single problem to solve), and is a

reasonable part of the Knowledge Sharing Pattern Language supporting the sharing of work status knowledge in a project team. Links from this pattern to other patterns were documented to a reference map (an excel table). All links at the pattern description were checked.

### 7.1.3 Validating Knowledge Sharing Patterns

Candidate patterns are analyzed in the validation process that will be better introduced later in Chapter 10. The result can be a validated pattern, a pattern requiring some rewriting or a pattern that cannot be validated and is rejected. Based on the sources used for developing a pattern, the validation of a pattern may differ. In this study, if a best practice is found more than once in practice and its results are proven successful, it can be accepted as a valid pattern (as will be shown in Section 10.2). A pattern can be directly validated, if it follows a widely recognized software development method. A pattern following an organization's processes can be validated if there is some kind of evidence for the use of the process in practice. Also, patterns introduced by others could be accepted if acceptable validation is introduced as part of the pattern documentation. If such a pattern must be significantly modified, the pattern must be separately validated.

The final validation will come through the use of a pattern in practice. One sign of validation is whether or not the people find the pattern useful in practice. This, however, takes much time, so in this study the validation has first been done at a more theoretical level. This kind of quick validation before wide introduction of patterns is important to ensure the appropriate quality of patterns.

### 7.1.4 Early Steps with the Resulting Knowledge Sharing Patterns

The creation of this set of knowledge sharing patterns has been an iterative process, like e.g. Vlissides (1998, p. 151) recommends. For the author, it has also been a learning process about how to write patterns. At a mental level this learning process has started many years ago. First pattern drafts were article writing patterns with which the author taught herself how to write scientific articles.

The first drafts of knowledge sharing patterns were introduced at the beginning of the year 2004. After that the KSF was developed and the idea of knowledge sharing pattern concept in general was further improved. It was apparent, after producing more knowledge sharing patterns that the pattern concept seemed to work even better than originally believed, which was very encouraging.

Creating the Knowledge Sharing Pattern Language has included several iterations. One of the biggest changes was implemented after a pattern writing workshop moderated by James O. Coplien and Juha Pärssinen (Tampere, Finland, March 2007). After the workshop the current names of the patterns were introduced and the patterns were looked through once again. Several patterns were split into two or more separate patterns. The result was a set of patterns that are much easier to understand and apply.

As discussed before in this thesis, the resulting patterns are still an early set of patterns, needing further improvement and real use. The development and validation of the resulting patterns have been made according to the process described in this Section 7.1. The main weaknesses are that the resulting patterns have still too many interdependencies and that the validation has not been possible in more than one real environment. Those patterns are, however, sufficient for the purposes of this thesis.

## 7.2 Summary of the Process for Creating the Knowledge Sharing Pattern Language

One of the contributions of this study is the process for creating and evaluating a pattern language, the Knowledge Sharing Pattern Language. The (iteratively implemented) steps of this process are:

1. Need emerges in the form of many practical problems having connections to difficulties in knowledge sharing.
2. Studying the domain, gaining better understanding of knowledge sharing.
3. Verifying the validity of patterns as a tool in this case – mapping with the knowledge sharing domain.
4. Defining goals – Knowledge Sharing Goal Tree.
5. Defining the structure based on Knowledge Sharing Goal Tree.
6. Developing knowledge sharing patterns.
7. Validating patterns.
8. Validating the pattern language.

This process was initiated based on practical problems in software engineering projects. After understanding that many of those problems had connections to difficulties in knowledge sharing, more understanding was required about the domain of knowledge sharing in software engineering. According to Winn and Calder (2002), a pattern is grounded in a domain (practice and theory). To develop a good pattern the domain should be well known.

The next step was to define a tool to improve knowledge sharing in software engineering. Patterns were selected as that tool because they supported most of the requirements for a tool to improve knowledge sharing in software engineering nicely. In addition, a knowledge sharing pattern format was defined and patterns created based on that. The Knowledge Sharing Goal Tree was guiding the decisions about how to structure the pattern language.

In parallel with the development of patterns, the validation of patterns was implemented. It has included the definition of how single patterns could be adequately validated before the long practical experience of use. After the validation of single patterns the whole pattern language was validated. Part IV (Evaluating) explains how the validation was implemented.

Based on these, a more general process for creating a pattern language could be defined. The steps are:

1. Recognition of a real life need
2. Understanding the domain

3. Decision: is there a need for a pattern language?
4. Selecting pattern format and mapping it to the domain
5. Defining goals for the pattern language
6. Defining the pattern language structure
7. Developing patterns
8. Validating patterns (parallel to the previous step)
9. Validating the pattern language

In order to have a real pattern language that produces coherent wholes (Alexander 1999), a real need for such a language is required. The best way to start this process is to base it on an identified practical need. Then, normally, more knowledge and study is required about the domain and the practice. The patterns, after all, should not be invented by the writer. They should be existing practices producing successful results.

After understanding the domain and the practices related to it better, considerations should be given to whether patterns and a pattern language really are tools helping to solve the problems. In some cases it could be enough simply to define one nice procedure to help all the situations or to introduce a new information system that does not require a whole new pattern language.

If patterns and a pattern language are potential solutions, the next step is to select the pattern format to be used and to define the targets for the pattern language. One possibility, which should work also at other domains, is the building of a goal tree and finding some reasonable structure for a pattern language while developing it. Based on those and relevant sources, the patterns need to be developed and validated. Finally the whole language needs to be validated.

This process has not been tested with other domains, but it should not be very difficult to utilize in practice. During this research this has been a very good process noticing that the development of patterns is an iterative process taking time. Thus also the validation part is iterative, and in many cases, like Winn and Calder (2002) state, the feelings must be related more than intellect. The real validation will come from the use of the patterns and the language, but before it, some first step of validation is required to ensure the potential of the patterns in practice.

# 8 USAGE OF PATTERNS

This chapter describes the main ways of using knowledge sharing patterns. First the four main ways of usage are introduced in Section 8.1. The following four sections each introduce one of these usages at more detail level.

## 8.1 Overview

There are four main ways of using knowledge sharing patterns (Figure 37, the letters in the figure refer to the following list):

(a) Using patterns to understand knowledge sharing
(b) Improving an organization's processes
(c) Identifying a risk or a problem and using a corrective pattern
(d) Situation dependent use

The main usages are introduced in the following sections. Those include also example scenarios of use including discussion how to find the patterns needed. A scenario (according to Clements et al. 2002, pp. 52-53) includes three parts. Those are: *a stimulus* explaining what the stakeholder does to initiate interaction, *an environment* describing what is going on at the time of the stimulus, and *a response* telling how the stimulus should be responded. In the next sections, these three parts are defined separately for each scenario and utilized in the following way:

- *Stimulus:* (imagined) concrete situation from practice.
- *Environment:* explaining what is going on at the time of the stimulus.
- *Response:* explaining how to apply and what are the knowledge sharing patterns.

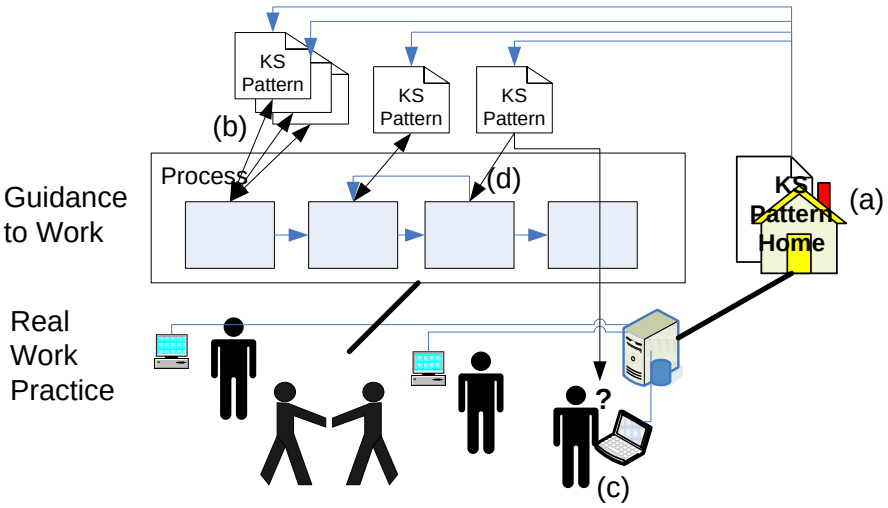



Figure 37. Knowledge Sharing Patterns in Use.

## 8.2 Using Patterns to Understand Knowledge Sharing

Knowledge Sharing Pattern Language is an entity that can be used as it is, to understand what knowledge sharing should or could mean in software engineering. It is built using html and every pattern is on its own web page that is easy to access to from other places or other patterns. A screenshot of the home page, the parent pattern *Improved Knowledge Sharing* (KSP00), is in Figure 38. For the full introduction of this pattern, see Appendix B.

The use of the Knowledge Sharing Pattern Language for understanding knowledge sharing could be initiated through the scenario introduced in Table 20. This scenario would require some kind of preliminary knowledge of the existence of knowledge sharing patterns.

To familiarize yourself with knowledge sharing in software engineering, the structure of the Knowledge Sharing Pattern Language could be used as a basis. The parent pattern KSP00 provides the initial understanding and links to the two views of knowledge sharing (knowledge sharing interface view and target knowledge stream view).

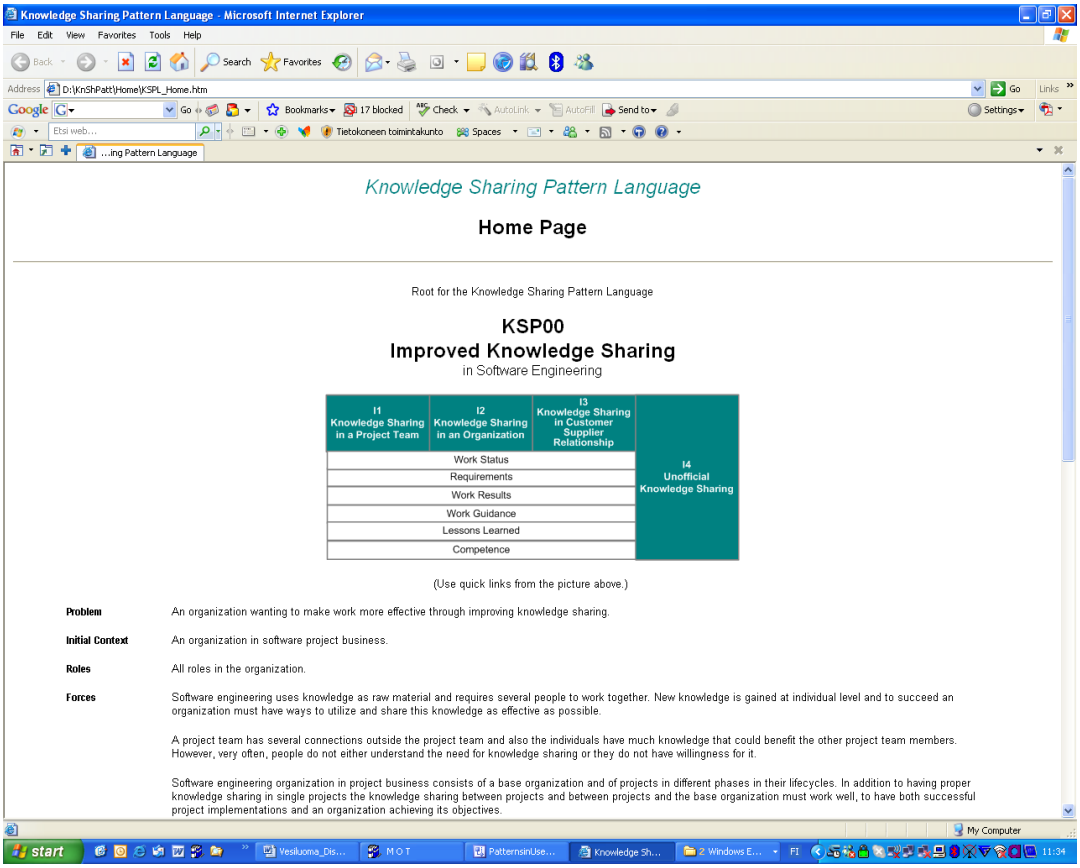



Figure 38. Screenshot from the home page of the Knowledge Sharing Pattern Language.

Table 20. Scenario for the Usage of the Knowledge Sharing Pattern Language for Understanding Knowledge Sharing.

| The Stimulus | The Environment | The Response |
|---|---|---|
| A person being responsible for project management process improvement in an organization cannot directly find a common factor among the problems in projects and starts to wonder if improving knowledge sharing could help. | Several kinds of problems in projects, for example repeating the same mistakes in different projects. | The person studies the Knowledge Sharing Pattern Language to find out how knowledge sharing could be supported and how it would affect the practice. Based on it, the person starts evaluating the current knowledge sharing situation in the organization and decides upon focus areas for improvement actions. |

## 8.3 Improving an Organization's Processes

The term *processes* is used here to refer to an organization's documented standard operating procedures. Those are very often the main content of a formal quality system of an organization. The knowledge sharing patterns can be used many ways to improve the processes of a software development organization. The processes of an organization can be improved based on the pattern solutions, those can be extended with patterns or patterns can be used to pilot new procedures before introducing them and including them into the organization's processes. The use of knowledge sharing patterns for improving existing processes of an organization is potentially the most common way of using knowledge sharing patterns.

The scenarios in Table 21 describe two example scenarios of improving an organization's processes with better knowledge sharing properties. As in the previous section, these scenarios require knowledge about the existence of knowledge sharing patterns. Through the coverage of the Knowledge Sharing Pattern Language it is very possible that there is knowledge sharing patterns for most of the required activities.

Table 21. Scenario for the Usage of the Knowledge Sharing Pattern Language for Improving an Organization's Processes.

| Id | The Stimulus | The Environment | The Response |
|---|---|---|---|
| 1 | An organization wanting to improve current processes with better knowledge sharing practices. | An organization whose existing processes that only support the work adequately. | Look through the target knowledge streams of the Knowledge Sharing Pattern Language. Identify the gaps and define and implement the improvements needed for the current processes. |
| 2 | An organization wanting to improve current processes with better knowledge sharing practices, but not wanting to replace current processes fully however. | | Look through the target knowledge streams of the Knowledge Sharing Pattern Language. Identify the gaps. Based on the identification add links to the relevant knowledge sharing patterns from the right places in an organization's process descriptions. For example, link to KSP04 Trust or Check pattern from the project management process and task assignments. |

Knowledge sharing patterns can be used to extend an organization's processes with better knowledge sharing features. Patterns can be used as they are, just linked from an organization's processes (scenario 2 in Table 21), or used when defining and building processes for an organization (scenario 1). Those can also be used as supplementary material when using processes. For example, from the project management process's progress follow-up phase there could be a link to the pattern *Trust or Check* (KSP04). This pattern could be supplementary material reminding of one potential problem area: how to know that a person is implementing his/her assignment as he/she should.

Knowledge sharing patterns could also be used to pilot new ways of doing things. For example the pattern *Shared Understanding* (KSP05) could have first been piloted in some projects and only after that would it be used to update the organization's processes with those practices. This is actually a combination of the scenarios in Table 21.

In general, with patterns, it is possible to add new features over existing processes, to have supplementary material as patterns and to pilot some practices later to be added as part of processes. Patterns could produce flexibility in companies having, e.g. ISO 9001 certified quality systems. Patterns can be introduced in addition to the formal quality system and give supporting details for implementing actions in software development projects. Patterns can also be used when designing processes.

With documented processes there is always a risk of over documenting. That could result in processes with which no-one wants to comply. Much guidance, however, is required nowadays to apply to many detailed customer requirements. Patterns could bring a way to have different additions over the processes. The whole system can be built so that the formal quality system includes the big picture of software development while the patterns give more detailed instructions to certain problem areas. For example, there could be different pattern extensions to processes for different customers. In such cases, the processes could be formed so as to give a more stabile structure while the patterns represent the flexible, more often changing part of the system.

Patterns could also be an excellent solution for the frequently asked questions (FAQ) lists. Very often the questions related to processes are questions that someone has already asked and for which solutions have been found. Because, many problems in software engineering have their origin in knowledge sharing problems, many patterns on this kind of a list could be knowledge sharing patterns.

## 8.4 Identifying a Risk or a Problem and Using a Corrective Pattern

Knowledge sharing patterns can be used to identify problem situations and to discover ways to help in such situations. Those knowledge sharing patterns can also be used as preventive solutions, when noticing a risk or some tricky problem.

Scenarios in this case are based on practical problems or risks in practice and will result in finding the right knowledge sharing patterns for the purpose. To define realistic scenarios, some

of the problem areas of project Alpha (Figure 15) are used here as examples of stimuli (Table 22). At Table 22 the italic font refers to knowledge sharing interface, the underline font refers to target knowledge type, and the bold font refers to other key words used when selecting the right knowledge sharing pattern.

Because of the consequential relations between the problem areas (see the arrows in Figure 15, page 35), the scenarios for problem areas P1-P5 are more reliable than the others. This is because those are causes for the other problem areas. To solve the other problem areas properly the causes need to be solved first. In Table 22 the problem areas P11-P15 are left out, because of that reason. This highlights the importance of proper analysis of problems at hand. When selecting knowledge sharing patterns, it is important to define the problem so that the cause is treated, not just the effect. For example P1 (weak project definition) has resulted in also P6 and P9, so the response to all of these is and needs to be very similar.

The logic of using the Knowledge Sharing Pattern Language is based on the analysis of the environmental part of the scenario. Based on it, the involved knowledge sharing interface and the target knowledge type are decided. These can then be used to select the supporting patterns in the case. Also, in some cases, the definition of the environment already includes clues referring to the pattern name. In Table 22 these references/clues are highlighted with different font types.

Table 22. Scenarios for Problem-Based Usage of the Knowledge Sharing Pattern Language.

| **The Stimulus** | **The Environment** | **The Response** |
|---|---|---|
| P1 Weak project definition:<br>Misunderstanding between the customer and the project manager about what has been agreed on to implement in the project. | The definition of the project does not include adequate requirements for the project and there is no **shared understanding** *between the customer and the supplier*. | Apply pattern KSP05 Shared Understanding to redefine, together with the customer, the results from the project and have common understanding about the project. |
| P2 Resourcing with inexperienced resources:<br>The project manager notices problems in project progress because a lack of competences in project team. | *The project team* does not have all the competences required for the project. | Apply pattern KSP18 Improved Competences to check the competency needs and to plan and implement required actions to have available required competences. |
| P3 Failure in providing organizational support:<br>When having problems, different persons from the organization tried to help in addition to their own assignments. No systematic and in some cases just taking time from the project, not really helping. | The *organization* does not have mechanisms *to support the project* with missing competences. | Apply pattern KSP01 Project Support to establish an expert network in the organization and to assign required experts or other support persons to this project. |
| P4 Failures on the organizational level for follow-up and control:<br>The organization noticing too late that the project has severe problems. | The status and problems of *project Alpha* became known too late in the *organization* to support the project efficiently. | Apply pattern KSP10 Known Status of Projects to have constant follow-up of projects in the organization. |
| P5 Problems in data communications | Problems with data network and connections. | Involve IT support and operators. (This problem was not directly the result of problems in knowledge sharing but produced those.) |
| P6 Disagreement between the company and the customer about what has been agreed on:<br>(An effect of problem area P1)<br>The customer evaluating the project results at a much higher quality level than the project team has understood being the requirement. | The understanding regarding the requirements of the project differs between the *project team* and the *customer*. | Apply pattern KSP05 Shared Understanding to recreate the shared understanding of project results. |
| P7 No jointly agreed on processes in use:<br>Problems in team work because of not following any special processes. People on the project team not knowing who is responsible for what. | **Processes** had not been defined for the *project* and the project did not utilize the organization's standard processes. | Apply pattern KSP22 Work Guidance to establish process discipline and to define processes that needs to be followed in this project. |
| P8 No proper verification and validation activities:<br>Project results produced in hurry are not proper for the purpose. | No systematic **procedures** in use for verification and validation in the *project*. | Apply pattern KSP22 Work Guidance to establish systematic basic verification and validation based on the organization's standard processes. |
| P9 Problems in customer communication:<br>(An effect of problem area P1)<br>Customer being angry and the communication being more antagonistic than cooperative to solve problems. | Because not having a common understanding about project requirements between the *customer and the supplier*, the communication did not work well at all. | Apply pattern KSP05 Shared Understanding to create the shared understanding. (See P1) |
| P10 Problems in project management:<br>(An effect of problem areas P1 and P2 – see also those)<br>Project basics not properly defined, not enough competence in project management to take over the project in the middle of a crisis. | The *project* is not managed well enough so that the basics for understanding the progress in project would be in place. | Apply pattern KSP09 Schedule Baseline to define the basics for project management. After that implement pattern KSP25 Followed Progress to know the status of the project compared to the baseline.<br>(Also KSP05 and KSP22) |

As an example of this logic, problem area P1 Weak Project Definition is used here. The environmental part of the scenario refers to the shared understanding which is actually a pattern name. The selection of a pattern name is very important and if it succeeds well, it can become a key word in practice, the pattern then is easy to find when it is needed. Without such direct mapping, the environmental part of the scenario needs to be studied more carefully in order to find the links to the structure of the Knowledge Sharing Pattern Language. In P1 the environment part of the scenario includes two parts referring to this. Based on "*between the customer and supplier*" the knowledge sharing interface is I3 (knowledge sharing in customer supplier relationship). Based on the word "requirement" the involved target knowledge type is Requirements knowledge. This creates pair I3-R which, in the Knowledge Sharing Pattern Language structure, refers to two potential knowledge sharing patterns: *Shared Understanding* (KSP05) and *Discovered Bones* (KSP06). Both patterns would help in this situation, but KSP05 utilizes also KSP06 and is better to start with in this case of problem P1. When a subset of knowledge sharing patterns (e.g. here for I3-R) is found, the pattern problems and contexts should be studied to find the right pattern or to verify that the pattern will help in the current situation.

In P3 Failure in Providing Organizational Support the resulting pair is I2-Comp (knowledge sharing in an organization and competence target knowledge type). This result in three knowledge sharing patterns: *Project Support* (KSP01), *Named Experts* (KSP02), and *Assigned Experts* (KSP03). Pattern KSP01 refers to the other two patterns being a parent pattern to those.

## 8.5 Situation Dependent Use

One way to use knowledge sharing patterns is to realize that in identified situations a certain pattern will be used and people are trained to recognize those situations. All knowledge sharing patterns are somehow situation dependent. Every knowledge sharing pattern has an initial context defining situations when those are applicable.

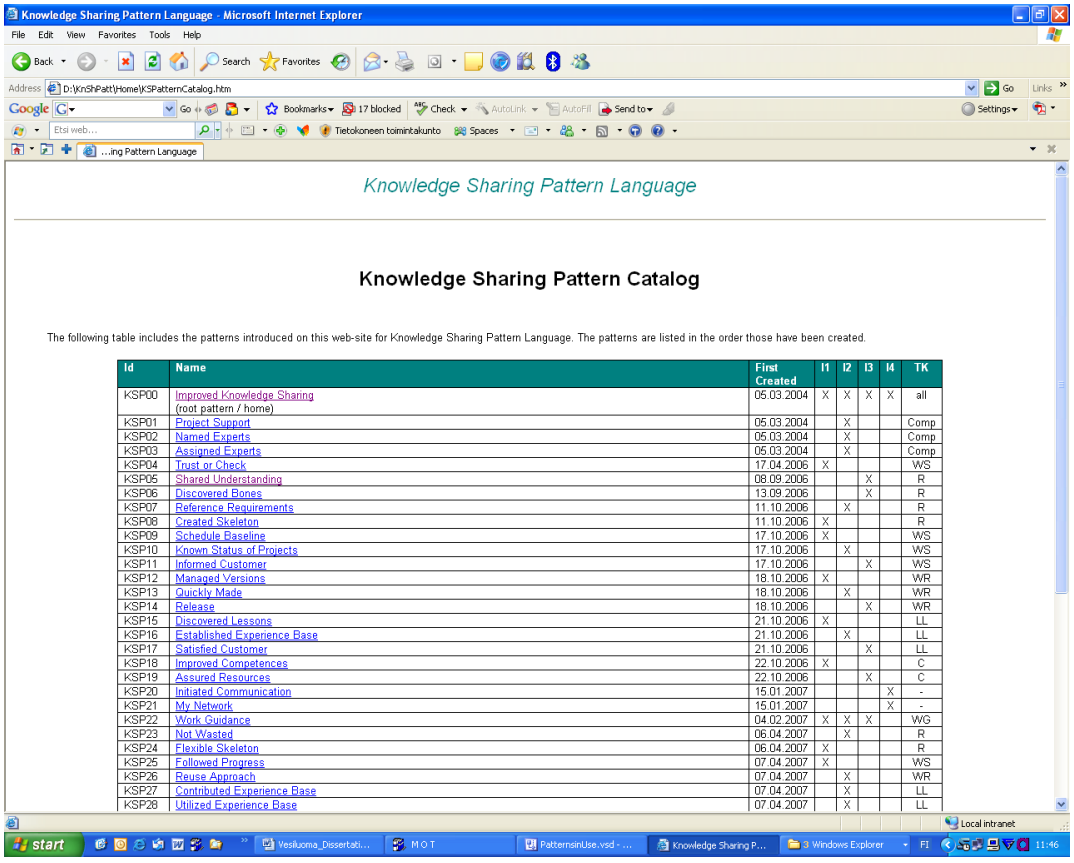
The example scenario would include a situation as the stimulus and the use of linked patterns as the response. Table 23 gives one example. There the situation is that a project is about to be closed and this raises the need to collect lessons that are learned from the project. This means that the actors need to learn this kind of behavior and to implement it always when closing a project. It could be started by first linking the knowledge sharing pattern *Discovered Lessons* (KSP15) from organizations procedures and stepwise training people to do that automatically without any external trigger in projects.

Table 23. An Example of Situation Dependent Scenario.

| The Stimulus | The Environment | The Response |
|---|---|---|
| The project manager notices that the project is approaching its conclusion. | Project is about to be closed | Use pattern KSP15 Discovered Lessons to collect the lessons learned from the project and to make those available to others in the organization. |

The pattern *Work Guidance* (KSP22) is an example of a pattern that is not very situation dependent. It is a pattern that should be used throughout the software development phases.

When knowing a certain pattern to look for, the pattern catalog page (Figure 39) of the Knowledge Sharing Pattern Language can be used. For full content of the page, see the Appendix B. At the bottom of the catalog page, there are also direct links to the knowledge sharing interfaces and target knowledge streams.




| Id | Name | First Created | I1 | I2 | I3 | I4 | TK |
|---|---|---|---|---|---|---|---|
| KSP00 | Improved Knowledge Sharing (root pattern / home) | 05.03.2004 | X | X | X | X | all |
| KSP01 | Project Support | 05.03.2004 | | X | | | Comp |
| KSP02 | Named Experts | 05.03.2004 | | X | | | Comp |
| KSP03 | Assigned Experts | 05.03.2004 | | X | | | Comp |
| KSP04 | Trust or Check | 17.04.2006 | X | | | | WS |
| KSP05 | Shared Understanding | 08.09.2006 | | | X | | R |
| KSP06 | Discovered Bones | 13.09.2006 | | | X | | R |
| KSP07 | Reference Requirements | 11.10.2006 | | X | | | R |
| KSP08 | Created Skeleton | 11.10.2006 | X | | | | R |
| KSP09 | Schedule Baseline | 17.10.2006 | X | | | | WS |
| KSP10 | Known Status of Projects | 17.10.2006 | | X | | | WS |
| KSP11 | Informed Customer | 17.10.2006 | | | X | | WS |
| KSP12 | Managed Versions | 18.10.2006 | X | | | | WR |
| KSP13 | Quickly Made | 18.10.2006 | | X | | | WR |
| KSP14 | Release | 18.10.2006 | | | X | | WR |
| KSP15 | Discovered Lessons | 21.10.2006 | X | | | | LL |
| KSP16 | Established Experience Base | 21.10.2006 | | X | | | LL |
| KSP17 | Satisfied Customer | 21.10.2006 | | | X | | LL |
| KSP18 | Improved Competences | 22.10.2006 | X | | | | C |
| KSP19 | Assured Resources | 22.10.2006 | | | X | | C |
| KSP20 | Initiated Communication | 15.01.2007 | | | | X | - |
| KSP21 | My Network | 15.01.2007 | | | | X | - |
| KSP22 | Work Guidance | 04.02.2007 | X | X | X | | WG |
| KSP23 | Not Wasted | 06.04.2007 | | X | | | R |
| KSP24 | Flexible Skeleton | 06.04.2007 | X | | | | R |
| KSP25 | Followed Progress | 07.04.2007 | X | | | | WS |
| KSP26 | Reuse Approach | 07.04.2007 | | X | | | WR |
| KSP27 | Contributed Experience Base | 07.04.2007 | | X | | | LL |
| KSP28 | Utilized Experience Base | 07.04.2007 | | X | | | LL |

Figure 39. The Catalog Page of the Knowledge Sharing Pattern Language.

# Part IV Evaluating

*This part describes the evaluation of the main results of this study. The main results are the Knowledge Sharing Framework from the understanding part of this study, and the Knowledge Sharing Pattern Language from the supporting part. Since the evaluation of the Knowledge Sharing Framework is done through the case studies introduced in Chapter 4, only a summary of the case studies is presented in Chapter 9. In Chapter 10, the evaluation of the Knowledge Sharing Pattern Language is discussed from several perspectives.*

# 9 EVALUATION OF THE KNOWLEDGE SHARING FRAMEWORK (KSF)

The actual evaluation of the KSF is carried out through industrial case studies (Chapter 4) where the use of the KSF resulted in many valuable findings. KSF helped to understand the problem areas and the potential best practices in the projects which were studied. Project Alpha (Section 4.1.2.2) had several problem areas. The interface between the customer and the supplier (I3) worked well in the beginning, but because of inexperience in project team (I1 interface) and lack of support from the organization (I2 interface) the project confronted several challenges that also resulted problems also in the customer supplier interface (I3). From the software engineering activity types, most of the challenges were in the area of managing software engineering (S1) and verifying software engineering (S3). During the project many of the problems were solved and by the end of the project (L3, project closure) the situation was also much better in the interface with the customer (I3).

Project Beta (Section 4.1.2.2) was a totally different story. There, not many real problems existed and potential reasons for that could be related to the potential best practices found in the customer supplier interface (I3) and in the interfaces inside the organization (I2). Also the managing software engineering activity type (S1) seemed to have many potential best practices that could be shared with others.

The use of the KSF helped to analyze both of the case study projects. By utilizing the KSF dimensions a certain analysis structure was achieved that focused the analysis on different important areas. Without such a structure, the analysis would be very superficial. KSF was effective also when evaluating the knowledge sharing approach of different software development methods (Section 4.2). KSF has proved its usefulness especially in finding difficulties or even a lack of knowledge sharing in cases, where it should have existed.

The results acquired from utilizing the KSF are presented as the KSF profiles (Figure 16, Figure 21, and Figure 22). A KSF profile proved to be a very efficient way of describing visually the findings and identifying the potential strengths and weaknesses of knowledge sharing in the target environments. Through the process of defining the KSF profiles for several projects in an organization, an understanding of potential common problems or best practices of knowledge sharing can be gained.

KSF is a simple framework that is easy to apply in practice, as the industrial cases demonstrate. Through its three dimensions (knowledge sharing interfaces, software engineering activity types and project life cycle phases) the KSF covers software engineering projects and their whole life cycle. The industrial case studies did not raise any need for additional dimensions. Actually, during the first case studies, it would have been enough to utilize only two of the three dimensions (knowledge sharing interfaces and software engineering activity types), because of the large amount of resulting data.

It might be possible to apply the KSF in projects other than just in software engineering. The software engineering activity types are quite general but the KSF might possibly apply as it

exists or after small modifications to the study of some other types of engineering projects. That, however, is not the purpose here.

# 10 EVALUATION OF THE KNOWLEDGE SHARING PATTERN LANGUAGE

The Knowledge Sharing Pattern Language is one of the main results of this study. In this chapter the technique used for evaluating the knowledge sharing patterns and the pattern language, as well as the results from the evaluation are introduced. An overview of the evaluation method of the Knowledge Sharing Pattern Language is given in Section 10.1. The evaluation consists of several steps, the evaluation of the individual patterns, the evaluation of the coverage of the pattern language, and the evaluation of the applicability of the pattern language, discussed in Sections 10.2, 10.3, and 10.4, respectively. Finally, in Section 10.5, the technique used for the evaluation is summarized and reflected in the evaluation of organizational patterns in general.

## 10.1 Evaluation Overview

A pattern language consists of several patterns which together, when applied in a reasonable way, should produce a coherent whole. To evaluate a pattern language the patterns need to be evaluated both separately and as a pattern language. The first section describes the approach used to evaluate single knowledge sharing patterns. The second section introduces the evaluation from the Knowledge Sharing Pattern Language perspective.

### 10.1.1 Pattern Level Evaluation

How to evaluate a pattern is not a trivial question. Patterns, before they are captured as a pattern description, are sort of the undocumented folklore of a community of knowing. They are the proven solutions for certain types of problems. The same pattern might be visible in different ways in different communities of knowing. Because it exists as a proven solution, the concept of a pattern already includes the assumption of the usefulness of the pattern in the intended context. Very often, definitions about what is a pattern are based on the existence of the problem-solution pair in practice. For example, Vlissides (1998, p. 147) says that when they produced the design patterns for the book Gamma et al. (1995) they had the rule of two existing examples of a problem and its solution before they would write a pattern for it. Thus, individual patterns should be justified in terms of existing applications, rather than being validated through new applications.

Organizational patterns could be compared to best practices. Wareham and Gerrits (1999) indicate that best practices are rarely validated instead they are based on the belief of the ‘best in class’ performing organizations as having excellent practices for others to follow. In contrast to design patterns, the existing instances of organizational patterns in software development processes are hard to recognize: organizational patterns are typically not visible in concrete

artifacts but in human interactions. They state also that instances of design patterns can be found even mechanically from the source code of a system. For organizational patterns this kind of 'source code' does not exist so first some kind of process modeling needs to be done.

The core of a pattern is the problem to be addressed by the pattern and the basic idea of the solution. Pattern instances are always applications of the pattern basic idea. Even in the case of design patterns, the example systems given as a justification of the pattern have actually not applied the pattern description itself (because the pattern did not even exist as a formal pattern description at the time), but the pattern idea. Thus, to justify a pattern it is sufficient to point out "successful" instances of the basic idea. In the case of organizational patterns the sources are project experiences, existing software development methods and models in the literature, in addition to well-founded observations in the literature.

Since the sources of organizational patterns are not as easily available as with design patterns, even one positive instance can be viewed as significant justification, provided that the context corresponds to that of the pattern. The evaluation can be further strengthened by providing a negative instance that is, an actual problem in an existing project that could have been avoided, had the team applied the pattern.

The individual knowledge sharing patterns are evaluated along these lines. Section 10.2 introduces the evaluation in general. The actual evaluation is introduced in Appendix B after each pattern description. For each pattern, both a positive and a negative instance are provided, if possible. In some cases a negative instance was not possible to identify, due to the nature of the pattern.

### 10.1.2 Pattern Language Level Evaluation

According to Alexander (1999) the "goodness" of a pattern language is based on its generativity, its capability to produce coherent wholes. Evaluating this is not an easy task, and, as Winn and Calder (2002) state, the evaluation is sometimes based more on feelings than intellect. One way to evaluate this is to analyze the coverage of the pattern language. The target for the Knowledge Sharing Pattern Language is to initiate and assure a good level of knowledge sharing in the software engineering work. Thus the coverage of the pattern language can be evaluated by analyzing to what extent the Knowledge Sharing Pattern Language covers a coherent whole of knowledge sharing in software engineering. This kind of evaluation is given in Section 10.3 using the Knowledge Sharing Framework. Essentially, this kind of evaluation provides evidence that the pattern language is sufficiently complete.

Another perspective for evaluating a pattern language would be its potential to be useful in practice. This is the user's viewpoint to a pattern language. In other words, given a practical situation in a project, does the pattern language provide solutions to cope with the related knowledge sharing problems? Neither the individual pattern evaluation (positive and negative instances) nor the coverage analysis provides a direct answer to this question. The former concentrates on the occurrences of a particular solution rather than on the situations needing this solution, and the latter considers problem areas rather than concrete problem situations.

Evaluating this kind of applicability of a pattern language is difficult in general. The evaluation problem is similar to the problem of evaluating the quality attributes of software architectures. For example, in order to evaluate the maintainability of a software architecture fully, one would have to imagine all the future, actual maintenance situations and analyze whether the architecture supports them or not. This is, of course, not possible. The solution proposed to this problem in the software architecture realm is to use scenario-based assessment, where the architecture is evaluated against a few representative concrete situations, rather than against general quality requirements. Naturally, the value of the evaluation depends on the choice of the scenarios, but if the scenarios represent the typical, most likely concrete situations, the evaluation provides significant evidence as to the quality of the architecture. To ensure that the scenarios represent all that they should, they can be arranged according to a predefined categorization (sometimes called a *profile*, Bosch 2000).

A pattern language is comparable to a software architecture. A pattern language can be regarded as the architecture of a system that assists a team member in a software development process. In the context of software architecture, a popular technique that is used to assess the quality of software architecture is to apply scenario-based approaches, like ATAM (Architecture Tradeoff Analysis Method, Clements et al. 2002). There relevant quality factors are selected and concretized by using scenarios. The resulting scenarios are then analyzed against the solutions in the architecture, identifying how those affect the realization of the scenario.

Following this line of thought, the relevant quality factors of a pattern language can be evaluated using the scenario-based Q-PAM method (Välimäki et al. 2008), similarly to software architectures. The applicability of the Knowledge Sharing Pattern Language is evaluated this way in Section 10.4. Note that the evaluation does not concern a possible tool (e.g. web pages) used to publish the pattern language, although quality factors, like usability, are of course relevant in such a tool, as well.

## 10.2 Evaluating Knowledge Sharing Patterns

The justification of basic ideas of single knowledge sharing patterns is given in Appendix B after every pattern description. The main sources for positive and negative instances have been practical project experiences and methods in literature. Mostly the positive instances originate from literature and negative instances originate from project Alpha. This also ensures that two different kinds of sources have been used. The patterns KSP16, KSP27 and KSP28 all have the same basic idea and evaluation. They all implement one part of the basic idea, which is to collect, store and share lessons learned systematically.

As an example, the justification of the pattern *Shared Understanding* (KSP05) is given here. This pattern was used as an example also in chapter 5.4. The evaluation includes first the basic idea and gives then a positive and a negative instance for that basic idea:

***Pattern Shared Understanding (KSP05) Basic Idea.*** Instead of just starting to define requirements for a project, here, the aim has been to decompose the project target into different

smaller parts. These are then negotiated with the customer in order to reach shared understanding.

***Positive Instance.*** This is a best practice that is found in an organization's project management process descriptions and in practice especially as a rule of three important aspects of defining a project: deliverables, acceptance criteria and change management. This practice together with one globally acting customer organization has also been confirmed. Coplien and Harrison (2005, p. 366) has defined patlet[5] *Shared Clear Vision* that has the similar purpose of creating first a shared vision about the system to be built. Also McCarthy and McCarthy (2006, pp. 11-20) highlight the importance of establishing a shared vision, or understanding as referenced here.

***Negative Instance***. In project Alpha one of the two main problems was problem P1: Weak Project Definition. This pattern would have helped, for example, through a more explicit acceptance criteria (including the quality requirements) that were missing in project Alpha during the first part of the project. Also the creation of the shared understanding would have been more systematic had this pattern been used.

## 10.3 Evaluation of the Knowledge Sharing Pattern Language Coverage

The coverage of the Knowledge Sharing Pattern Language is partially based on the coverage of the KSF, which was piloted in an industrial case study and was found to be an efficient tool for making the knowledge sharing visible in software engineering (Chapter 4). Based on the KSF, a Knowledge Sharing Goal Tree (see Section 5.1) was made. This Knowledge Sharing Goal Tree was utilized when structuring and planning the Knowledge Sharing Pattern Language. By using this kind of a structure, the coverage of the Knowledge Sharing Pattern Language was ensured during the construction phase. The resulting distribution of patterns compared with respect to the Knowledge Sharing Goal Tree structure is introduced in Table 24. The table has two columns on the left based on the Knowledge Sharing Goal Tree. A column for existing knowledge sharing patterns is added to the right.

As can be seen from Table 24, the resulting set of knowledge sharing patterns covers all target knowledge type and knowledge interface (I1-I3) pairs. For example, *Shared Understanding* (KSP05) focuses on the pair I3 and *requirements* type of knowledge. The aim is not to cover the unofficial knowledge sharing (I4) very effectively. Only two example patterns (KSP20 and KSP21) are defined for it. Covering every imaginable knowledge sharing situation in software engineering work would not be possible nor reasonable. The coverage of knowledge sharing patters is quite extensive, however, as can be seen from Table 24. The Knowledge Sharing Pattern Language has the potential to give a good understanding of knowledge sharing in software engineering work.

---

[5] A pattern in patlet form. "A patlet is a terse summary of a pattern's problem and solution." (Coplien and Harris, 2005, p. 30).

Table 24. Knowledge Sharing Pattern Coverage.

| TK Type | Interface | Knowledge Sharing Patterns |
|---|---|---|
| | | KSP09 Schedule Baseline |
| | | KSP25 Followed Progress |
| | I1 | KSP04 Trust or Check |
| | I2 | KSP10 Known Status of Projects |
| **WS** | I3 | KSP11 Informed Customer |
| | | KSP08 Created Skeleton |
| | I1 | KSP24 Flexible Skeleton |
| | | KSP07 Reference Requirements |
| | I2 | KSP23 Not Wasted |
| | | KSP05 Shared Understanding |
| **Reqs** | I3 | KSP06 Discovered Bones |
| | I1 | KSP12 Managed Versions |
| | | KSP26 Reuse Approach |
| | I2 | KSP13 Quickly Made |
| **WR** | I3 | KSP14 Release |
| | I1 | |
| | I2 | |
| **WG** | I3 | KSP22 Work Guidance |
| | I1 | KSP15 Discovered Lessons |
| | | KSP16 Established Experience Base |
| | | KSP27 Contributed Experience Base |
| | I2 | KSP28 Utilized Experience Base |
| **LL** | I3 | KSP17 Satisfied Customer |
| | I1 | KSP18 Improved Competences |
| | | KSP01 Project Support |
| | | KSP02 Named Experts |
| | I2 | KSP03 Assigned Experts |
| **Comp** | I3 | KSP19 Assured Resources |

During the mining of the knowledge sharing patterns, first, an early set of eleven patterns was created. Then the Knowledge Sharing Goal Tree was utilized to see how the knowledge sharing patterns would populate it. In the Knowledge Sharing Goal Tree, a place for a pattern was found by analyzing which knowledge sharing interface and what target knowledge type it concerned. In the early set there were two patterns for which it was not easy to locate a place in this tree. Those were found to be too general for real use so they were refined and positioned to the tree.

After locating a place for the early set of knowledge sharing patterns, the Knowledge Sharing Goal Tree is used to find missing coverage. The next step was to start mining patterns related to the missing knowledge sharing interface and target knowledge-type streams. Eleven knowledge sharing patterns were mined based on this need resulting in twenty-two knowledge sharing patterns that covered the Knowledge Sharing Goal Tree quite well. Later, six of these twenty-two knowledge sharing patterns were found to be difficult to give a proper name. At the same time we noticed that they do not have the scope and modularity required for the knowledge

sharing patterns. Those six patterns were split into two patterns each resulting in the final twenty-eight knowledge sharing patterns.

The limitation of this coverage evaluation is that the same structure used for developing the patterns is used for evaluation. With this coverage evaluation we can show that the resulting patterns are populating pretty well the Knowledge Sharing Goal Tree structure. This does not show what possible elements might be missing from the Knowledge Sharing Goal Tree and thus from the coverage evaluation. The Knowledge Sharing Goal Tree, however, is based on the KSF coverage that is found good in the industrial study.

## 10.4 Evaluating the Applicability of the Knowledge Sharing Pattern Language

As discussed in Section 10.1.2, the applicability of the Knowledge Sharing Pattern Language and its limits are evaluated utilizing a scenario–based assessment. The Q-PAM evaluation method used is introduced in Section 10.4.1 and after it the implemented evaluation workshop in Section 10.4.2. Finally, the conclusions are given based on the evaluation workshop results in Section 10.4.3.

### 10.4.1 The Q-PAM Method

The Q-PAM method (Quality-oriented Process Assessment Method, Välimäki et al. 2008) follows ATAM-like (Clements et al. 2002) approach evaluating the quality of a process pattern language[6] through scenarios. Q-PAM method was created for evaluating processes or process patterns. Originally it includes four phases: a) *Creating a quality profile*, b) *Refining the quality profile*, c) *Constructing scenarios*, and d) *Analysis*. Phase b) *Refining the quality profile* was noticed not to be necessary in this case.

The quality profile can be created many ways. In this study, as in Välimäki et al. (2008), the external and internal quality factors defined in ISO 9126 (ISO 2001) are used as the basis. It includes two levels of quality factors (Figure 40). The first level (e.g. Functionality) is refined with more detail quality factors (e.g. Suitability).

[6] The Knowledge Sharing Pattern Language consists of organizational patterns, but those are normally from the perspective of processes so those can be called process patterns as discussed in Chapter 2.3.1.

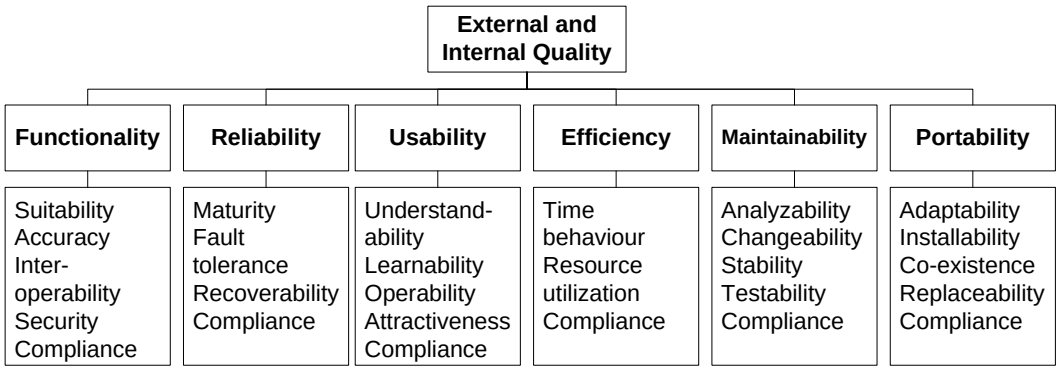



Figure 40. ISO 9126 External and Internal Quality Factors (ISO 2001).

The quality factors in ISO 9126 are defined for software products. Those are, however, working very well also in this study after some minor modifications to the explanations in ISO 9126. In most of the cases it is enough to replace the term *software product* with the term *Knowledge Sharing Pattern Language*. See the explanations for the selected quality factors in the Table 25. This works, because a pattern language can be thought to be a system, like a software product is a system. Both need to have certain quality factors included when designing, implementing and evaluating. Also, as discussed earlier, a pattern language needs to produce a coherent whole, like a system should also be.

Because these quality factors from the ISO 9126 standard worked well in pilot runs (Välimäki et al. 2008), those have also been used in this study. We are not claiming that these are perfect, we just claim that those work well in this kind of environment also, and are thus used in this study. Some other alternatives could also been considered.

A group of selected quality factors form a quality profile that is used in the evaluation. After deciding upon this quality profile, the scenarios are constructed for each quality factor. The scenarios are like test cases that can be run against the process patterns. These describe a specific requirement related to the desired quality properties of the process. Usually, a scenario can be expressed as a concrete situation occurring in an imaginary project, but in some cases it is more appropriate to give a scenario as a refined quality requirement rather than as an example. Essentially, a scenario in this context has three basic characteristics: it must be typical and frequently occurring, it must be directly related to knowledge sharing, and it must be precise enough to be used for evaluating the process. The latter property may be sometimes achieved only through iteration.

During the analysis each scenario is analyzed against the process patterns. A tag, describing the evaluation result, explanation, and the involved patterns are defined for each scenario. The tag has in this study been one of the following: supporting, some support or no support. Tag supporting means that the pattern language includes elements that support the realization of the scenario. Some support means that there are some elements supporting the scenario, but those are either not enough or there is not enough information to define that the scenario is fully supported. No support means that the pattern language does not have any identified support for the scenario.

If the tag is supporting or some support, the explanation needs to state clearly why the patterns support the realization of the scenario. The explanation needs to refer to the patterns ensuring this support. The patterns referred to in the explanation are summarized in the list of involved patterns.

Originally, in the Q-Pam method (Välimäki et al. 2008), the tags were similar to ones very often used in e.g. CMMI (Chrissis et al. 2003) assessments: F = fully passing the scenario test, L = largely passing the scenario test, P = partially passing the scenario test and N = not passing the scenario test. In this study, however, it was not possible to use these, because process patterns are quite abstract and it is difficult to collect evidence that a scenario is "fully passing". Because of this, the tags have described whether or not the pattern language supports the existence of the scenarios.

## 10.4.2 Evaluation Workshop

An evaluation workshop was arranged for the evaluation of the Knowledge Sharing Pattern Language. Four faculty members from the Tampere University of Technology were invited to participate along with the author. Three of the participants did not have any prior knowledge about the Knowledge Sharing Pattern Language. They were the main actors in this workshop. All participants had prior experience using the ATAM method in industrial context.

The workshop was arranged in two half-day sessions. The first session included a brief introduction to the purpose of the Knowledge Sharing Pattern Language and the Q-PAM method. The candidate-quality profile was introduced, discussed and agreed upon. The reminder of the first session was used for defining the scenarios. After the first session, the author made a proposal for the analysis. It was examined in the second session scenario by scenario and corrected according to the findings in the workshop. The following sections introduce the phases better.

### 10.4.2.1 Creating a Quality Profile

The author introduced a candidate-quality profile in the first evaluation session. It was accepted after some discussion. Because of the limited time available, only a subset of quality factors from the ISO 9126 external and internal quality factors were used. Reliability and maintainability related quality factors (see Figure 40) were left out, because those are not critical in the case of the Knowledge Sharing Pattern Language. Similarly the compliance (standards, conventions etc.) was left out of each quality factor area. The rest of the unselected quality factors were left out because of their very low importance in this case. The selected quality factors forming the quality profile used in this evaluation workshop are introduced in Table 25. Also the explanations of these quality factors are introduced in the same table. Together these quality factors form the quality profile that is used as the basis in this evaluation.

Table 25. The Quality Profile Used in the Evaluation. The explanations are modified from ISO 9126.

| Quality Factor | Explanation |
|---|---|
| **Suitability (Functionality)** | The capability of the Knowledge Sharing Pattern Language to provide an appropriate set of phases for specified tasks and user objectives. |
| **Accuracy (Functionality)** | The capability of the Knowledge Sharing Pattern Language to provide the right or agreed results or effects with the needed degree of precision. |
| **Interoperability (Functionality)** | The capability of the Knowledge Sharing Pattern Language to interact with one or more specified processes. |
| **Understandability (Usability)** | The capability of the Knowledge Sharing Pattern Language to enable the user to understand whether it is suitable, and how it can be used for particular tasks and conditions of use. |
| **Time Behaviour (Efficiency)** | The capability of the Knowledge Sharing Pattern Language to provide appropriate response and processing times and throughput rates when performing its tasks, under stated conditions. |
| **Resource Utilization (Efficiency)** | The capability of the Knowledge Sharing Pattern Language to use appropriate amounts and types of resources when the process performs its tasks under stated conditions. |
| **Adaptability (Portability)** | The capability of the Knowledge Sharing Pattern Language to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the process considered. |

#### 10.4.2.2 Constructing Scenarios

In the context of software architecture evaluation, Clements et al. (2002, p. 52) define a scenario as "a short statement describing an interaction of one of the stakeholders with the system." Accordingly, in this study a scenario is a short statement describing a concrete situation in a software development process. The situation has two basic requirements: first, it must be typical and frequently occurring, and second, it must be directly related to knowledge sharing.

The main part of the first workshop session was used for constructing the scenarios. The scenarios were decided by the three participants who did not have any previous knowledge about the Knowledge Sharing Pattern Language. This way the independence of the scenarios with the Knowledge Sharing Pattern Language was ensured.

One part of the discussions was about the level of the scenarios and how the Knowledge Sharing Pattern Language was aimed to be used. The knowledge sharing patterns affect the project normally through processes (see Figure 41). For example, a pattern is utilized when defining an organization's standard process. Later, a project utilizes the standard process and through that also implements the pattern. Another option for utilizing a pattern is that a project uses directly (not through existing processes) a knowledge sharing pattern suitable for the current situation (the gray arrow in Figure 41).

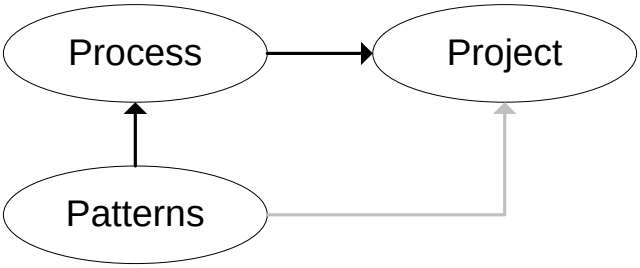



Figure 41. Linking the Patterns, a Process and a Project.

This could result in different ways of evaluating the knowledge sharing pattern language. It was decided that there are two main levels that are looked at in this case. They are the meta-level and the process-level. The meta-level refers to the features of the Knowledge Sharing Pattern Language as a language and their effect on processes. The process-level refers then to the practical project situations, the instances of the single patterns in real life.

The workshop participants looked through each selected quality factor of the quality profile. It was noticed that the quality factor *suitability* (from the group of functionality) was present in most of the scenarios defined. Similarly, all scenarios had the potential to serve several quality factors at the same time. Thus it was decided that it is not important which quality factors are served in which scenarios. More concern was taken to ensure that there are scenarios serving analyses of all the relevant quality factors. After going through the quality factors, some thoughts were given to find what kind of typical scenarios would still be missing. Four new scenarios were added based on this examination.

The result was thirty-one scenarios, twenty-three of which were process-level scenarios and eight meta-level scenarios. Originally there were thirty-four scenarios, but three of those were found to be either duplicates or refinements of the other scenarios. The resulting scenarios are introduced in the next section.

#### 10.4.2.3 Analysis

The resulting scenarios and the analysis per each scenario are introduced in the Appendix C. One example is given in this section. For each scenario we have given a scenario id (e.g. Scenario 1), the type of the scenario in parenthesis (meta-level or process-level), and the scenario itself after these. The main quality factor tells the quality factor which is considered when defining the scenario. In the parenthesis is the group to which this quality factor belongs in the ISO 9126 standard. Identified quality factor is not the only quality factor that the scenario serves. The result tells the tag: supporting, some support or no support. Explanation gives the reasoning for the selected tag. Finally, the involved patterns are listed.

***Scenario 1*** (Process): A new version management tool is needed in a project. The training will be arranged ten days prior to the use.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: Supporting

***Explanation***: Knowledge Sharing Pattern Language supports the identification of competences required for a project (KSP19, assuming that the tools have been remembered as one competence area) and the planning of required trainings for a project (KSP18). KSP18 is originally defined to be used only in the beginning of a project, but here it is assumed to be utilized also during project realization. Introduction of a new tool may change the software development processes in use (KSP22).

***Involved Patterns***:

- KSP19 *Assured Resources*
- KSP18 *Improved Competences*
- KSP22 *Work Guidance*

## 10.4.3 Workshop Conclusions

### 10.4.3.1 Result Based on the Tag Values and Involved Patterns

At the total level, from the thirty-one scenarios, fourteen gets support from the Knowledge Sharing Pattern Language, twelve gets some support, and five scenarios are not supported. For some of the scenarios more evidence or more thorough study would be needed before defining those supporting. This is the case especially with the three meta-level scenarios (7, 14 and 26), where the Knowledge Sharing Pattern Language gives at least some support. The other five meta-level scenarios get full support from the Knowledge Sharing Pattern Language. Thus, at the meta-level, meaning the pattern language as a whole, we could say that the Knowledge Sharing Pattern Language gives strong support to the defined scenarios.

At the process-level eighteen of the twenty-three scenarios get at least some support from the knowledge sharing patterns. Nine (39 % of the scenarios) of those are fully supported by the knowledge sharing patterns and eight of those get some support. Five scenarios do not get any support from the Knowledge Sharing Pattern Language. At the process-level 39 % of the scenarios are fully supported and 78 % of the scenarios get either full or some support. Based on this the Knowledge Sharing Pattern Language seem to give quite strong support to the separately-defined scenarios and is therefore important in supporting such positive scenarios.

When examining which knowledge sharing patterns were not referred to in the explanations, there was only one such pattern, *Project Support* (KSP01). It, however, fully consists of two other patterns (*Named Experts* KSP02 and *Assigned Experts* KSP03) that were referred to and thus involved. Based on this, we should define that the whole Knowledge Sharing Pattern Language was found to be needed in these scenarios, and all patterns were found reasonable. In most of the scenarios, by nature, the unofficial knowledge sharing related patterns (*Initiated Communication* KSP20 and *My Network* KSP21) were not relevant.

When ignoring those scenarios where involved patterns referred to all patterns or similar, the knowledge sharing pattern *Work Guidance* (KSP22) was referred to in nine scenarios. The next most frequently referred to pattern was *Discovered Bones* (KSP06) with eight references. Patterns *Shared Understanding* (KSP05) and *Managed Versions* (KSP12) got six references and *Schedule Baseline* (KSP12) got five references. These knowledge sharing patterns supported many of the scenarios and could thus be thought to be important patterns in the pattern language.

During the evaluation one of the evaluators, not having previous experience of the Knowledge Sharing Pattern Language, browsed the knowledge sharing patterns in their web-based environment and commented that it was quite easy to find patterns there and to use them there. He also found several patterns that supported the scenarios which the author did not notice herself before the second session.

#### 10.4.3.2 Limitations of the Knowledge Sharing Pattern Language

During the evaluation, several improvement possibilities to the knowledge sharing patterns were found. Here those are discussed briefly in the form of limitations of the Knowledge Sharing Pattern Language.

One of the findings was that there are some knowledge sharing patterns that are meant to be used primarily in the beginning of a project (L1 project establishment), but they could be very useful, as well, during a project, when there are bigger changes. Such patterns would need an extension to the Initial Context so that the potential user would understand this possible use. Examples of such knowledge sharing patterns are *Schedule Baseline* (KSP09) and *Improved Competences* (KSP18) (see Scenario1).

The Knowledge Sharing Pattern Language does not provide very good support for the achievement of the right quality level of documentation, communication and other related issues. As a result of the scenario analysis, support was missing for the identification of the right level of detail with the requirements (Scenario2 and Scenario4), not questioning what the customer defines as requirements (Scenario3), not supporting with the needed quality of documentation (Scenario5), neither understanding the potential of possible language/term misunderstandings in the communication (Scenario6). Similarly the language does not give much support for the identification of the right people to interview in the customer organization when requirements are being defined (Scenario28), even though the patterns remind of the need to identify the right stakeholders.

The knowledge sharing patterns include references to risk management in general, but the escalation-type of communication of risks/defects is not included (Scenario8 and Scenario18). Paying better attention to risk management would improve the Knowledge Sharing Pattern Language.

The Knowledge Sharing Pattern language should be used to improve the software development processes in an organization, but the language does not support the improvement of processes by defining what kind of a process is clear and easy to understand (Scenario12). *Work Guidance* (KSP22) pattern describes process management in general, but does not support process management with quality requirements for processes. It neither supports nor serves as a reminder of the importance of defining organizational standards for e.g. the use of tools (Scenario22). KSP22 is a very general pattern so it could be understood to cover these types of topics, but it is not really serving as a reminder of the importance of tools and the use of tools as part of the processes.

Reviewing is included in the form as a necessity for a review for requirements (KSP06). There is not, however, much real support for reviewing and motivating importance of reviewing as found in the analysis for Scenario16.

In the analysis of Scenario20 and Scenario29 the need for better support for establishing a project memory, a systematic approach for storing of project related information is raised. Some support is received from *Managed Versions* (KSP12) and *Work Guidance* (KSP22), but those do not systematically cover this area. Another, closely related limitation is that the Knowledge Sharing Pattern Language does not give much support for how to find results from the earlier projects so they may be reused in new projects (Scenario21). The pattern language includes general support for reusing but is not really much help in how to find some piece of earlier–produced, reusable work results.

In addition to the limitations, several improvement ideas for the Knowledge Sharing Pattern Language were raised during the workshop sessions. For example, it was identified that the Knowledge Sharing Pattern Language supports situations where a project team grows in numbers (Scenario24) or the amount of project team members is radically decreased (Scenario25), but there could be some more guidance about how to apply the knowledge sharing patterns in these situations. There could be one or two patterns referring to other knowledge sharing patterns and instructions on how to use them on these occasions. Another example is that the use of the KSF (Scenario13) could be one new knowledge sharing pattern supporting the identification of knowledge sharing challenges before starting to implement other patterns in practice and processes.

During the workshop a thought was raised that the knowledge sharing patterns could include better introductions of the pre- and post- conditions for knowledge sharing. These could either be as part of the Initial Context and Resulting Context or they could be a separate topic in the pattern format that is used. This would also bring more pattern language type of approach such as references to other patterns in the patterns of the Knowledge Sharing Pattern Language.

#### 10.4.3.3 Value of this Evaluation

The value of this evaluation depends very much on the choice of the scenarios. By using people who do not have prior knowledge about the Knowledge Sharing Pattern Language, the independence of the evaluators has been granted. In practice this means that the evaluators have defined the scenarios based on the knowledge of the purpose of the Knowledge Sharing Pattern Language and some examples of the language. They have had the idea of the Knowledge Sharing Pattern Language in their mind but do not know the implementation well.

The coverage of the scenarios is not and cannot be “complete”. Utilizing their experience, the evaluators have tried to find out the most typical, potential-use scenarios in knowledge sharing in software engineering projects and organizations. The use of the quality profile has supported the achievement of good enough coverage but has not limited the work. The four last scenarios have been considered after going through the quality factors in the quality profile. The aim at that point was to still think about some other most typical situations that were not yet covered properly.

One of the limitations of this evaluation is that there was not enough time to look through every detail of the quality of single patterns. This cannot fully replace a real-time use of the patterns, but it can give good enough understanding about the potentials of the Knowledge

Sharing Pattern Language to support actual life situations. Also this is a quick way to evaluate a pattern language before real training and use of people.

As a summary, the results of the evaluation were very encouraging. Most of the scenarios defined by people who did not have any previous experience with the Knowledge Sharing Pattern Language, were either fully or partially supported. Only five scenarios of the thirty-one were not supported at all. The evaluation also resulted in important knowledge about the limitations of the Knowledge Sharing Pattern Language being at the same time good raw material for further improvements of the pattern language.

## 10.5 Technique for Evaluating Organizational Pattern Languages

The technique for evaluating the Knowledge Sharing Pattern Language includes three parts: evaluation of single patterns, evaluation of the coverage of the pattern language, and evaluation of the applicability of the pattern language. Knowledge Sharing Pattern Language is an organizational/process pattern language, and this technique for evaluation can be utilized also for other organizational pattern languages. This evaluation scheme is briefly discussed in the sequel from the perspective of how to generalize this evaluation method for organizational/process pattern languages.

The evaluation of single patterns is based on defining the basic idea of the pattern solution, and then justifying it with positive and negative instance. This technique is based on evaluating the existence of the pattern in real life and should be directly applicable to any organizational pattern language.

The coverage of a pattern language is in this study mostly based on actions taken during the construction of a pattern language. The evaluation is then more like noticing that the intended structure is followed. In this study, the structure used to ensure the coverage is the Knowledge Sharing Goal Tree. It is based on two critical process perspectives: knowledge sharing interfaces and target knowledge types. For another organizational pattern language, the coverage can be studied in a similar way by identifying first two or more basic process dimensions that are important in the case. The selection of the process dimensions must be well established and reasoned. In this study, those are a result of defining the KSF based on the literature and then further defining the target knowledge types based on the software engineering activity type. Also the reasoning behind the KSF could be used as an example when defining the process dimensions for an organizational pattern language.

Finally, the applicability of the Knowledge Sharing Pattern Language is evaluated in an evaluation workshop that was based on practical scenarios. This is directly applicable with other organizational pattern languages and would highlight also the need of thinking about how the pattern language should be used. The evaluation workshop made the limitations of the pattern language clearer and resulted in many improvement ideas.

When utilizing this evaluation technique, several possible problems were identified and improvements initiated. Evaluating organizational pattern languages before real use is always

challenging. Later on this evaluation technique could be extended with an addition of one more part: a survey to be implemented after introduction and a short period of use of a pattern language. The target could then be to study which patterns have been used and which patterns have not. It could also study whether new versions of patterns have been initiated and what have been the main changes. This could not be implemented during this study but it could be a further improvement later. It would bring useful knowledge when improvements are considered for this process of creating a pattern language.

# Part V Closure

*This part summarizes the study and positions it into the research field. It includes the comparison of the results of this study to related work in Chapter 11, and other discussion including the review of the research question, research method and contributions in Chapter 12. Chapter 13 introduces the conclusions.*

# 11 RELATED WORK

This chapter includes a review to related work. It is divided into two parts: knowledge sharing in software engineering in general (Section 11.1), and the pattern approach to support knowledge sharing in software engineering (Section11.2).

## 11.1 Knowledge Sharing in Software Engineering

To study the related work regarding to knowledge sharing in software engineering, first, a grouping is made based on the literature. An introduction describes the grouping. After that, the sections follow the grouping to discuss related work. Finally, a short summary is given.

### 11.1.1 Introduction

Based on the literature, years 2002 and 2003 have been starting points for wider discussions on knowledge management in the context of software engineering (knowledge sharing being a subgroup of it). In 2002 IEEE Software journal had a special section on knowledge management in software engineering. The article introducing the area was Rus and Lindvall's (2002) article *Knowledge Management in Software Engineering*. That article has been referenced several times in this thesis and used frequently by other authors too (e.g, citations in ISI Web of Knowledge, Web of Science Citing). The other articles in that issue covered topics like knowledge management in organizations (Liebowitz 2002, Ramasubramanian and Jagadeesan 2002), learning from earlier projects (Birk et al. 2002, Schneider et al. 2002, and Komi-Sirviö et al. 2002), process management (Ramesh 2002), and knowledge management tools (Wei et al. 2002).

The book Managing Software Engineering Knowledge (Aurum et al. 2003) was published in 2003. When using a similar grouping as above, the articles of the book could be divided into: knowledge management in organizations (Edwards 2003, Lindvall and Rus 2003, and Dybå 2003), learning from earlier projects (Oliver et al. 2003, Rosenberg 2003, van Solingen et al. 2003, van Solingen 2003, Ebert et al. 2003), process management (Verner and Evanco 2003, Ebert et al. 2003, Jalote 2003), knowledge management tools (Dingsoyr and Conradi 2003, Shepperd 2003, Vegas et al. 2003), and other application areas, like requirements management (Dutoit and Paech 2003, Lowe 2003) and competence management (Althoff and Pfahl 2003).

Most of the results of this study are related to the group *knowledge management in organizations.* KSF and the Knowledge Sharing Pattern Language can be categorized in such a way as to belong to this general group. Single knowledge sharing patterns cover several different knowledge sharing application areas. Knowledge sharing patterns are quite general, so in many cases the related work can deepen the insight into the topic of a single knowledge sharing pattern.

At next sections the grouping introduced above is used for structuring the introduction of related work.

## 11.1.2 Knowledge Management in Organizations

This general category of knowledge management in organizations is can be divided into sub-categories based on different perspectives. First, the role of knowledge management in software engineering is introduced based on one source and the knowledge sharing patterns mapped according to it. After that, knowledge sharing models, knowledge sharing interfaces, and communities of practice, are briefly discussed based on the related work.

### 11.1.2.1 Role of Knowledge Management in Software Engineering

According to Rus and Lindvall (2002, pp. 31-32) the role of knowledge management in software engineering is: to support core software engineering activities (e.g. document management, competence management, expert identification and software reuse), to support product and project memory (e.g. version control, change management, documenting design decisions and requirements traceability), and to supporting learning and improvement (e.g. predictive models and qualitative project information like case-based systems). These roles have here been used when evaluating the coverage of the Knowledge Sharing Pattern Language. Table 26 summarizes the mapping of these roles with the knowledge sharing patterns.

Knowledge sharing is a sub-group of activities in knowledge management, but all examples given by Rus and Lindvall (2002) are somehow applicable and related to knowledge sharing. As can be seen from Table 26, the Knowledge Sharing Pattern language has patterns to support all these three main role categories. The first category, support for core software engineering activities has, and also needs to have more patterns mapped to it. The other two categories also have a good amount of patterns mapped to and supporting those roles.

At the *support core software engineering activities* role, when looking at the examples given by Rus and Lindvall (2002), the document management could not be directly identified from the knowledge sharing patterns. Expert identification and software reuse were well covered. Competence management was adequately covered, but missing some basic competence management related patterns. However, that is more the need from the general knowledge management perspective, not necessary from the knowledge sharing perspective.

Missing document management is an important finding. It became apparent when looking through the knowledge sharing approach of selected software development methods (Section 4.2). Two of those three methods were very much based on artifact-based knowledge sharing. A large amount of those artifacts are different kinds of documents. Support for document management would be required to manage those artifacts properly. The main reason for missing this is the use of target knowledge type streams. Those are good in support of certain types of knowledge flows, but they are missing the horizontal perspective covering all streams, like document management does. This will be one improvement need for the next version of the Knowledge Sharing Pattern Language.

Table 26. Mapping Roles of Knowledge Management in Software Engineering by Rus and Lindvall (2002) with the Knowledge Sharing Patterns.

| Roles by Rus and Lindvall (2002) | Knowledge Sharing Patterns (per identified sub area) |
|---|---|
| **Support core software engineering activities** | General: KSP01 Project Support, KSP05 Shared Understanding, KSP14 Release, KSP22 Work Guidance<br>Competence management: KSP18 Improved Competences, KSP19 Assured Resources<br>Expert identification: KSP02 Named Experts, KSP03 Assigned Experts<br>Software reuse: KSP07 Reference Requirements, KSP13 Quickly Made, KSP23 Not Wasted, KSP26 Reuse Approach |
| **Support product and project memory** | Version control: KSP12 Managed Versions<br>Change Management: KSP06 Discovered Bones, KSP09 Schedule Baseline, KSP11 Informed Customer, KSP24 Flexible Skeleton<br>Documenting design decisions: not properly included.<br>Requirements traceability: KSP08 Created Skeleton |
| **Support learning and improvement** | Lessons learned: KSP16 Established Experience Base, KSP15 Discovered Lessons, KSP27 Contributed Experience Base, KSP28 Utilized Experience Base<br>Improvement actions: KSP17 Satisfied Customer |

In the *support product and project memory* role, the knowledge sharing patterns mostly cover the introduced examples (version control, change management, documenting design decisions and requirements traceability). The main improvement need comes from missing support to encourage documenting the design decisions, and, as mentioned earlier in this thesis, from the need of improving the pattern *Managed Versions* (KSP12). The patterns do not yet support version management as well as it should. Most probably the pattern KSP12 will need to be split into two or more separate patterns to serve better in this purpose.

In the *support learning and improvement* role, qualitative project information is well covered. The experience base that would be required can be implemented e.g. as a case based system, which was mentioned as an example by Rus and Lindvall (2002). Another example was predictive models. Those are not necessary for knowledge sharing, but they could be, like measuring information in general, an efficient tool to support knowledge sharing. Measuring type information could be categorized to belong to the lessons learned target knowledge stream being input for lessons learned. The measuring type of input might be more effectively included in the next version of the Knowledge Sharing Pattern Language.

There were also five patterns that could not be listed in the mapping table. Those were three project progress follow-up patterns (*Trust or Check* KSP04, *Known Status of Projects* KSP10, and *Followed Progress* KSP25) and the two unofficial knowledge sharing patterns (*Initiated Communication* KSP20, and *My Network* KSP21). To support knowledge sharing in software engineering, the project progress follow-up is important, but here it is not included into the core software engineering activities. The unofficial knowledge sharing is not easy and perhaps it is not even relevant to include it into any categorization.

Dingsøyr et al. (2009) have studied different approaches to knowledge management and what practical implications there could be for software engineering practice. They have utilized the knowledge management methods or approaches identified by Earl (2001). As a result, they recommend practitioners to think about different approaches in organizations to knowledge management based on the software development approach. According to them, traditional software development will benefit more the technocratic schools approaches, and agile development more from behavioral schools. Earl (2001) defines technocratic schools being based on information or management technologies supporting employees in their everyday tasks. The technocratic schools approaches are more reactive compared to the behavioral schools which, according to Earl (2001), are stimulating mangers to be proactive in the creation, sharing and use of knowledge as a resource. In general this is similar finding as when utilizing KSF to study different software development approaches.

#### 11.1.2.2 Knowledge Sharing Models

The literature of the field includes different kinds of knowledge sharing models. The approach in these resources varies widely. The knowledge sharing model presented by Nonaka and Takeuchi's (1995) has been used widely (e.g by. May and Taylor 2003, Lindvall et al. 2003). It is introduced based on Nonaka et al. (2006) in Section 2.1.1. There knowledge is created and shared through knowledge conversions between tacit and explicit knowledge. It works on a different level than the KSF. In the KSF the target is to identify knowledge sharing situations when Nonaka et al. (2006) identify different types of knowledge sharing.

One very different approach to knowledge sharing is the knowledge flow network approach. For example Zhuge (2006) has the perspective of managing knowledge flows in teams to ensure knowledge sharing. Compared to the KSF it is intended more for defining optimal, required knowledge flows, whereas the KSF identifies the current situation and improvement needs.

Oliver et al. (2003) introduce a software engineering knowledge-sharing (SEKS) model. Based on prior experience, motivation and supportive culture, SEKS results in the ability to share knowledge. The SEKS model does not overlap with the Knowledge Sharing Framework (KSF). It defines the prerequisites to have knowledge sharing while the KSF studies the existing knowledge sharing or identifies the lack of it. Several articles from this group are utilized throughout this thesis to explain the knowledge sharing environment and its requirements.

#### 11.1.2.3 Knowledge Sharing Interfaces

The idea of using knowledge sharing interfaces as a basis when studying knowledge sharing is supported also by other authors. In this study, the main interfaces (Figure 10) are defined to be: inside a project team, inside an organization, and between organizations (customer and supplier relationship). As an example, Bosch-Sijtsema (2004) has defined a knowledge transfer framework for virtual projects. She has defined the knowledge transfer levels: intra-project knowledge transfer, inter-organizational knowledge transfer, service partner knowledge transfer, and inter-project knowledge transfer. These support the selection of knowledge sharing interfaces in the KSF. Intraproject knowledge transfer means the same as the knowledge sharing interface inside a project team. The inter-organizational knowledge transfer is towards alliance

partners in the virtual project. In the context of this study, this could be viewed as an intra-organizational knowledge transfer inside an organization. Bosch-Sijtsema's inter-project knowledge transfer would also belong to this category (knowledge transfer in an organization and especially between projects). Finally, the service partner knowledge transfer would correspond to the knowledge sharing interface between organizations, e.g. between the supplier and the customer.

There are also other kinds of interfaces. For example, Nakakoji (2006) refers to interfaces: the relationship of an individual with artifacts, the relationship of an individual with other developers, and the relationship of an individual to the community as a whole. The interfaces in the KSF are mostly relationships between developers (knowledge sharing interfaces in a project team and in an organization), but the unofficial knowledge sharing includes also relationships to the other developer community. Nakakoji's interfaces do not cover knowledge sharing with the base organization or with externals, especially with customers. KSF then does not cover the relationship between an individual and an artifact as a separate interface. This relationship may be in place in some knowledge sharing situations, for example, when an artifact is used as a tool in sharing knowledge, but the actual interface is then between individuals.

#### 11.1.2.4 Communities of Practice

Mestad et al. (2007) have studied communities of practice in the context of a software consulting company. They have studied different models on supporting communities of practice in a medium-sized software consulting company. Through the time span of four years there were three different types of models used in the company. The support for communities of practice started with evening seminars continued with special interest groups and finally was replaced with skill circles. The main difference between these models was the amount of flexibility and involvement of participants. Those both grew through these models. In the Knowledge Sharing Pattern Language, in pattern *Named Experts* (KSP02), one of the actions is the initiation of sub-communities. In addition, pattern *Initiated Communication* (KSP20), includes support to similar kinds of activities. This article, and in general research regarding how to support communities of practice in software engineering, are in line with the defined knowledge sharing patterns. This Mestad et al. (2007) article could be used to bring new experience-based knowledge to improve single patterns (e.g. the mentioned ones) of the Knowledge Sharing Pattern Language.

### 11.1.3 Learning from Earlier Projects

Learning from earlier projects and experiences is an important part of knowledge sharing and ensuring that the same mistakes are not repeated. On a more general level this approach is called the learning organization approach and it is applied more and more in software engineering. Schneider et al (2002) introduce their experiences in implementing a learning software organization, and van Solingen et al. (2003) introduce how to facilitate learning through control loops in software engineering. Learning can take place in different ways, for example through learning from already existing repositories (see Oliver et al. 2003), through measuring and the resulting data analysis (see van Solingen 2003), through lessons learned or a postmortem

analysis (see Rosenberg 2003 and Birk et al. 2002), or through an experience factory (see Basili et al. 1994) based approaches (see Komi-Sirviö et al. 2002). In addition, there are several articles reporting different types of lessons learned from projects (see Damian 2007, lessons learned from requirements management) and some studies regarding to lessons learned systems (see Andrade et al. 2007) and using project postmortem databases (see Schalken et al. 2006).

One important group of knowledge sharing patterns is related to learning from earlier projects: the knowledge sharing patterns representing the lessons learned target knowledge stream. Those patterns describe this area of learning at very general level including identifying lessons learned from projects (*Discovered Lessons* KSP15), utilizing those in organization (*Established Experience Base* KSP16, *Contributed Experience Base* KSP27, and *Utilized Experience Base* KSP28), and learning from the customer relationship through understanding how the customer perceives the project (*Satisfied Customer* KSP17). Clearly, this is an area, where many more knowledge sharing patterns could exist (e.g. more effort on measuring based lessons learned), and where some more detailed examples could benefit the existing knowledge sharing patterns. However, the defined knowledge sharing patterns already have a good start in covering this area.

### 11.1.4 Process Management

The group process management as part of knowledge management grouping includes several kinds of approaches. Processes itself are a storage of knowledge in an organization, and software process improvement (for example Zahran 1998) introduces several topics related to that. One approach being also part of software process improvement approach is the measuring based approach of CMMI (Chrissis et al. 2003). Process management includes several different approaches that intend to improve knowledge sharing. One example would be to improve support for traceability (following the life of developed objects) through process management (Ramesh 2002).

Process management is not a focus area in this study. As shown earlier in this thesis, patterns can support process management by extending processes with new features (e.g. with knowledge sharing features, see Section 8.3). The work guidance target knowledge stream including pattern *Work Guidance* (KSP22) is part of the Knowledge Sharing Pattern Language. It gives some general guidance for establishing process support for software engineering work.

### 11.1.5 Knowledge Management Tools

One branch of discussion related to knowledge sharing in software engineering is related to the tools that are used for knowledge management and especially for knowledge sharing. Current tools, techniques and methodologies are inadequate to address knowledge management effectively in software engineering (Ward and Aurum 2004). The contributions of this study are intended to support especially methodologies. No specific tool for managing an experience base is selected and studied. As noted by Desouza (2003), there are several barriers to the effective use of knowledge management systems in software engineering. Desouza also reminds us that

these systems are only one means to foster knowledge. Petter et al. (2007) remind us that several tools exist to support knowledge sharing, not just the knowledge-base type of related tools. Introducing some tool types as examples in the knowledge sharing patterns could bring additional value to the knowledge sharing patterns. Lindvall et al. (2003, p. 139) states that "a collection of technologies for authoring, indexing, classifying, storing, contextualizing and retrieving information, as well as for collaboration and application of knowledge" are required to support knowledge management. Koskinen et al. (2004) give an example of utilizing hypertext support in sharing knowledge in an organization to the software maintainers.

As mentioned, the Knowledge Sharing Pattern Language does not directly refer to any knowledge management tool. It has links, however, to that discussion, through the lessons learned knowledge stream and especially to the experience-base patterns (KSP16, KSP27 and KSP28). Those do not define any specific tool to be used but indicates that some kind of storage/storages will be required.

### 11.1.6 Other Application Areas

One application area of knowledge sharing in software engineering has been the sharing of architectural knowledge. The origin of this area has been much earlier but, for example, a separate workshop was initiated for that in June 2006, Torino, Italy. A discussion was held there about tailoring knowledge sharing to the architectural process (Farenhorst 2006). After that, there have been several other studies, related to perquisites of sharing architectural knowledge (Farenhorst et al. 2007), architectural knowledge management in general (Babar and Gorton 2007), and patterns about how to organize architectural planning in different size of projects (Booch 2008).

In the Knowledge Sharing Pattern Language architectural knowledge sharing is part of the work results target knowledge stream. Architectural knowledge is not separately covered in Knowledge Sharing Pattern Language, but as one type of the work results. One possible extension of the Knowledge Sharing Pattern Language could be to take better notice of the importance of the architecture and architectural knowledge in a project.

Virtual teams and distributed software engineering gives new challenges to knowledge sharing in software engineering. There is a rich body of literature that has grown up around them. Topics cover many areas, for example team knowledge and coordination (Espinosa et al. 2007), project management in virtual teams (Casey and Richardson 2006), communication practices (Paasivirta 2005), and knowledge coordination (Kanawattanachai and Yoo 2007). Distributed software engineering is an important approach because software engineering work is more and more done in global virtual teams or by several teams located round the globe. In this study this virtual/distributed perspective is left out for simplicity, but based on this discussion much could be learned also about knowledge sharing in co-located teams. Distance between project team members makes lack of knowledge sharing much more visible.

Agile approaches to software development have gained more maturity during the past few years. As noticed in Section 4.2.2, agile methods (XP used as an example), when compared to traditional methods, have more direct knowledge sharing through communication and

collaboration between people. Of course, it must be noticed that strong direct communication alone does not ensure knowledge sharing. Also e.g. reflection and time would be needed to create knowledge based on the communication.

Traditional approaches are more artifacts centric in formal knowledge sharing (Chau et al. 2003). Studying agile approaches is important in studying and supporting knowledge sharing in software engineering. Chau and Maurer (2004) have continued studying agile methods and knowledge sharing, and so have many others. For example Karni and Kaner (2005) have studied agile knowledge-based decision making. Actually, nearly all agile related articles and books refer to knowledge sharing.

In addition to an agile approach, a product line approach (see Sugumaran et al. 2006) also has many connections to knowledge sharing. As discussed e.g. in Section 4.2, product line-based methods are a systematic way of reusing earlier work results. For example, the knowledge sharing pattern *Reuse Approach* (KSP26) refers to a product line approach. As with agile methods, product line based methods also have the potential for extending the Knowledge Sharing Pattern language.

KSF is built to allow for the study of knowledge sharing in different kinds of software development approaches, for example, in agile and product line-based approach. Similarly, the Knowledge Sharing Pattern Language allows for agile or other software development approaches. Agile and product line-based approaches are used as a source of examples in the Knowledge Sharing Pattern Language. For example, the knowledge sharing pattern *Discovered Bones* (KSP06) refers in practice examples to agile approaches.

### 11.1.7 Summarizing

Knowledge management, including knowledge sharing, has gained interest in software engineering during last years. The general discussion has links to the Knowledge Sharing Framework and the structure of the Knowledge Sharing Pattern Language. The application areas of knowledge sharing research in software engineering then have connection points to the single knowledge sharing patterns. The Knowledge Sharing Pattern Language could be thought to be a general structure which could be extended with new knowledge sharing patterns introducing better and different knowledge sharing application areas.

To focus the study, certain related areas have been consciously left with less attention. In particular, organizational learning (e.g. Cohen and Sproull 1996) is a related research area with voluminous literature. Although organizational learning is not in the focus of the thesis, it has inspired several of the research referred to, for example Mestad et al. (2007).

## 11.2 Pattern Approach to Support Knowledge Sharing

### 11.2.1 Patterns and Software Engineering

From the general approach to knowledge sharing in software engineering the focus is next changed to a pattern approach. IEEE Software, Volume 24, Issue 4, had a special section on software patterns. There Kirchner and Völter (2007, p. 28) stated that "patterns have become part of the software development mainstream," and "they're available for all software development phases". Even though this is the case, according to Buschmann et al. (2007) for many software engineers the Gamma et al. (1995) book of design patterns is still very influential and they are unaware of the maturity of the other field of software patterns.

Domains and technologies documented by patterns are according to Buschmann et al. (2007): distributed computing, language- and domain-specific idioms, fault tolerance and management, security, embedded systems, process and organizational structures, and education. Knowledge Sharing Pattern Language belongs to the group *process and organizational structures*.

### 11.2.2 Organizational Patterns

Organizational patterns, or according to Buschmann et al. (2007) grouping, process and organizational structures patterns, are one type of patterns. Several sets of organizational patterns and/or pattern languages exist; two of those have briefly been discussed in next paragraphs. Some of the resulting knowledge sharing patterns have the field *Releated Patterns* with appropriate references to existing patterns, but more mappings would be possible. In this section some examples of related patterns and pattern languages are discussed. In addition to the two pattern languages / sets of patterns introduced in this section, also e.g. Coplien's earlier patterns (Coplien 1995) available in Internet and the patterns collected by Schuler (2008) have been referred to in the *Related Patterns* fields in patterns in Appendix B.

Manns and Rising (2005) have defined patterns for driving and sustaining change in an organization. The resulting set of patterns includes 48 patterns and has patterns to support every stage of a change process. The patterns have been defined over several years starting from the year 1996. Those have been shepherded and workshopped at several workshops and conferences, for example, at several Pattern Languages of Programs Conferences (PLoP). According to the authors (Manns and Rising 2005, p. xvii): "At each of these conferences and workshops, participants improved the patterns by sharing their experience and suggesting new patterns." The patterns are a result of years of documenting, observing, collecting information and feedback from people who are introducing new ideas etc. The patterns are not reflecting ideas of the authors but those of many people round the world.

Some of the patterns introduced in the Manns and Rising (2005) pattern language support the patterns in the Knowledge Sharing Pattern Language. For example, very often, when a software engineering project starts, they often start with new ideas with which the project team is not familiar. Pattern Mentor (Mann and Rising 2005, pp. 192-194) could help in such a situation. The summary of that pattern is: "When a project wants to get started with the new idea, have

someone around who understands it and can help the team." This is very similar to the knowledge sharing pattern *Assigned Experts* (KSP03), where, in the beginning of a project possible competence gaps are identified and experts assigned. Manns and Rising (2005) pattern Mentor is referred to in the pattern *Assigned Experts* (KSP03) description at field *Related Patterns* (see Appendix B).

Coplien and Harrison's (2005) Organizational Patterns of Agile Software Development actually include four pattern languages: Project Management, Piecemeal Growth of the Organization, Organizational Style, and People and Code. The first two pattern languages lay the foundation of the software engineering organization and the last two "deal with the nuts and bolts of creating the thing" as Coplien and Harrison (2005, p. 29) define it. According to them, the patterns are not invented, but they are waiting to be discovered and documented. The data gathering for these patterns is made through several interviews and social network analysis with approximately 100 organizations. After different analyses, common patterns have been observed from that material. Those have been captured in pattern form, links between patterns have been identified and the sequences organized into pattern languages.

Coplien and Harrison's Project Management pattern language (Coplien and Harrison 2005, pp. 31-98) covers the organizational aspects of managing projects. It has, for example, connections with the work status target knowledge stream patterns in the Knowledge Sharing Pattern Language. For example, Coplien and Harrison's pattern Work Split could be used as an example of how to define tasks in the knowledge sharing pattern *Schedule Baseline* (KSP09). Another example could be a potential extension to the competence target knowledge stream in the Knowledge Sharing Pattern Language. It is Coplien and Harrison's pattern Day Care. There an expert is put in charge of all novices in a project, and letting the others develop the system.

The Piecemeal Growth of the Organization pattern language (Coplien and Harrison 2005, pp. 99-175) defines the ways in which an organization grows and develops over time. It offers patterns to strengthen and fine-tune an organization using feedback and insight. For example, Coplien and Harrison's patterns Engage Customer (coupling the Customer role with the Developer and Architect roles) and Surrogate Customer (role in project team for a person who will try to think like a customer) could extend the Knowledge Sharing Pattern Language by improving knowledge sharing in the customer supplier relationship.

The Organizational Style pattern language (Coplien and Harrison 2005, pp. 176-234) defines the general approach to the way the organization works. Several patterns there refer to roles and responsibilities in a project and in an organization. In addition there are some patterns supporting communication. Those include, for example, the pattern Face to Face Before Working Remotely, the pattern Shaping Circulation Realms (creating organizational structures encouraging communication) and The Watercooler pattern (encouraging social structures unrelated to workplace structures). The last one has a very similar basic idea as in the knowledge sharing pattern *Initiated Communication* (KSP20).

The last one of the four pattern languages of Coplien and Harrison (2005, pp. 235-284) is the People and Code pattern language. It defines the ways in which people affect code and the ways the design of code affects people. From the perspective of the Knowledge Sharing Pattern Language, People and Code pattern language deals mostly with topics related to the work results target stream. The detail level in Coplien and Harrison's pattern language is higher than it is in

the Knowledge Sharing Pattern Language. People and Code pattern language has also some patterns dealing with sub-teams and teams in different locations. At this phase distributed software development has not been a focus area in the Knowledge Sharing Pattern Language.

Regarding the future of organizational patterns, Buschmann et al. (2007, p. 36) say: "The growing adoption of agile development processes suggests that the corresponding pattern literature will continue to grow. Some work will focus on macroprocess aspects, such as overall life-cycle and business interaction, and some will focus on microprocess aspects, such as test driven development, refactoring, and tool use."

Agile approaches are not the main focus area in the Knowledge Sharing Pattern Language. It is important, however, as one of the mainstream software development approaches and material for single patterns and examples are looked at and will be looked at as well from agile methods. The growing amount of new, agile software development patterns will be especially interesting when improving the Knowledge Sharing Pattern Language.

Knowledge Sharing Pattern Language represents the macroprocess aspects, introducing the richness of knowledge sharing in software engineering. Single knowledge sharing patterns could be defined to represent microprocess aspects. Those, however, are not detailed enough for that level. Linda Rising (2007, p. 46) has stated that "determining the appropriate level of abstraction is an old debate in the patterns community". In this study, it is not reasonable to try to reach adequate microprocess level in sub-areas of the Knowledge Sharing Pattern Language. Instead, it would be good to extend the Knowledge Sharing Pattern Language with patterns or pattern languages made by others and to cover the microprocess level that way. Then the Knowledge Sharing Pattern Language could act as an index for some microprocess level pattern languages. Knowledge Sharing Pattern Language could help in structuring pattern languages that support knowledge sharing in software engineering in this way. The problem of how to organize patterns is important and is discussed e.g. by Welicki et al. (2006) and Hafiz et al. (2007).

### 11.2.3 Knowledge Sharing Patterns

Knowledge management and especially knowledge sharing in software engineering are growing areas for introducing patterns. Very often patterns are very practical by nature with relatively few research results available. Knowledge sharing is widely discussed but support is not yet available through patterns.

The term *pattern* is used in discussions related to knowledge sharing, but very often in a more general sense (see Ardichvili et al. 2006), not as the structured pattern mentioned here. Patterns are also implemented by utilizing the pattern format as the storing structure in knowledge bases (see Persson et al. 2003).

Persson and Stirna (2007) have introduced patterns for a knowledge management approach adoption. There the patterns are used similarly as here in the Knowledge Sharing Pattern Language, but their focus is more knowledge base centric. Strohmaier and Lindstaedt (2005) have defined knowledge problem patterns and they use those to identify the lack of knowledge sharing. Compared to the KSF those are even more generic. The way of introducing the

Knowledge Problem Patterns through a knowledge process picture is interesting and it could be evaluated if it can be utilized to support the introduction of the findings when using the KSF.

Zhuge (2006) has defined a set of knowledge flow patterns aimed at establishing an effective modeling approach for the management of knowledge flow in teams. Zhuge's patterns do not follow the pattern format used here. They are different models of knowledge flow networks. They do have a similar purpose to that of knowledge sharing patterns in that the goal is to provide efficient knowledge sharing. The main difference between the two, however, is that the knowledge flow patterns are general in presentation without examples of concrete knowledge sharing situations which the knowledge sharing patterns provide.

Paasivaara (2005) has found a set of eighteen successful communication practices. Because communication is also an important part of knowledge sharing these communication practices are very interesting. Paasivaara has introduced the communication practices for inter-organizational product development. She has utilized a pattern format when describing these communication practices. Many of her practices could be, after minor modifications, included into the knowledge sharing patterns. An example could be the communication practice Visiting Engineer. Paasivaara (2005, p. 156) describes it in the following way: "Visiting engineers visit the collaboration partner; customer, subcontractor or subsidiary, and stay and work there for a longer period of time. They facilitate communication by passing information, creating contacts, solving problems, and simply by being present for face-to-face discussions." This could be a knowledge sharing pattern for the knowledge sharing interface I3 (customer supplier relationship). The target knowledge type could be all target knowledge types. As part of the knowledge sharing pattern introduction of this pattern, different examples could be given for different target knowledge types. These communication practices could be studied as candidates for the next version of the Knowledge Sharing Pattern Language. They also show that improving knowledge sharing, including communication, is very important especially in software engineering projects.

Schuler (2008) has published *Liberating Voices!* pattern language for communication revolution. The patterns concentrate on communication "as a crucial arena in the battle for equality and justice" (Schuler 2008, p. 1). Those are mostly out of scope from the perspective of this study, but some of the patterns were found to support also knowledge sharing patterns. For example the pattern Thinking Communities supports the implementation of knowledge sharing pattern *Initiated Communication* (KSP20) encouraging to worldwide networking.

It must be noticed that several already existing pattern languages have patterns that could, after quick restructuring, serve as knowledge sharing patterns even as part of this Knowledge Sharing Pattern Language. Some examples are given in Section 11.2.2.

### 11.2.4 Summarizing

Since 1995 and the design patterns of Gamma et al. (1995) patterns have arrived to software engineering in many ways and for many purposes. Also organizational patterns, like knowledge sharing patterns are, have found their place in software engineering. Knowledge Sharing Pattern Language introduces the knowledge sharing domain at a macroprocess level and, yet, it could be

extended through external patterns or pattern languages for different knowledge sharing application areas.

Patterns and pattern languages in the context of knowledge management, knowledge sharing and communication start to emerge. In addition to extending the Knowledge Sharing Pattern Language with these, patterns from other organizational pattern languages for software engineering also could be utilized to extend the microprocess level of the Knowledge Sharing Pattern Language.

# 12 REVIEWING THE RESEARCH

This chapter binds together the elements of this study by first reviewing the research questions in Section 12.1, and then describing how the contributions of this study provide answers for these in Section 12.2. Also the research method that was used is discussed in Section 12.3, as well as the limitations of the results in Section 12.4.

## 12.1 Reviewing the Research Question

This study highlights the importance of knowledge sharing in software engineering. This need arises from real life software engineering challenges. Software engineering is knowledge work made by people. A software engineering organization very often has required knowledge somewhere, but not necessarily available when a single project needs it. This is the result of a lack of knowledge sharing in different interfaces between people. A person might be doing good work on his or her project but if the project manager forgot to explain that there was a change that affected the module the person is developing, how can he or she create the best possible results? In many cases these things do not happen because people would like to harm your work. These take place because of a variety of circumstances and situations such as being too busy, dealing with very complicated matters, not knowing the possibilities of the organization, and human mistakes among many others.

A result of the several challenges with knowledge sharing in software engineering is the need to find out ways to support knowledge sharing. Before we are able to support it, however, our target for this support first needs to be well understood. Thus, the research question was formulated in the following way:

*How to understand and support knowledge sharing in software engineering?*

The research question was then opened up to two sub-question areas and finally to six sub-questions. The understanding part included three sub-questions:

1. How to reach understanding in stakeholder interfaces and activities and when does knowledge sharing take place?
2. How is knowledge sharing visible in software engineering?
3. How can knowledge-sharing situations be represented?

The supporting part included also three sub-questions:

4. How may knowledge-sharing problems and solutions be represented?
5. How can knowledge-sharing situations in software engineering be supported?
6. How can we assure sufficiently-wide coverage of knowledge-sharing solutions?

The answers to the research question including the sub-questions are made explicit by the contributions of this study. These are reviewed in the following section.

## 12.2 Reviewing the Contributions

The contributions are discussed here according to the research sub-questions and by adding a couple of contributions that did not have a direct connection to the sub-questions. This section is divided into three sections, understanding, supporting and evaluation. See also Table 1 for the summary of research sub-questions and contributions.

### 12.2.1 Contributions – Understanding

The first research sub-question related to understanding was: *How to reach understanding in stakeholder interfaces and activities and when does knowledge sharing take place?* The answer for this and for the resulting contribution is the Knowledge Sharing Framework (KSF). It is also the main result from the understanding part. It is a tool for identifying situations where knowledge sharing should take place. By studying such situations, the knowledge sharing status in certain environments can be made visible and possibilities can be found for improvement.

KSF is based on three dimensions: knowledge sharing interfaces, software engineering activity types and project life cycle phases. With these dimensions, the research sub-question can be answered and the environment studied for knowledge sharing situations. Evidence of the results of using this framework is given in Chapter 4 where it is applied in case studies and for studying different software development methods. Several findings of real life knowledge sharing problems and potential best practices were the result of the use of the KSF. These findings were used as input for improvement planning in the organization being studied.

The second research sub-question was: *How is knowledge sharing visible in software engineering?* The answer is twofold: the KSF profiles created based on the results from analysis with the KSF, and the results from the comparison of selected software development methods.

In addition to having the KSF as a tool for analyzing, the ways of sharing the results are important. KSF profiles visualize efficiently the analysis results making it possible to compare and combine results of different projects. Based on the results of single projects, a KSF profile can be created having the common or potentially common elements from the single projects summarized. As a result of the comparison of selected software development methods, the main strengths and weaknesses of those methods become clear. It is important to know them in order to prepare other activities to support the weak parts or to utilize this knowledge when selecting the right approach for a project or for an organization.

The third sub-research question was: *How can knowledge-sharing situations be represented?* The answer for that is the knowledge sharing domain model. One of the main findings when creating and utilizing KSF was that knowledge sharing cannot be separated from its real environment. If knowledge sharing is examined only as a knowledge flow between two roles, the context affecting the knowledge sharing is left out and the motivation and understanding of the situation is not included.

### 12.2.2 Contributions – Supporting

The fourth research sub-question and first one related to supporting was: *How may knowledge-sharing problems and solutions be represented?* The answer is the knowledge sharing pattern concept. It includes a certain pattern format used for knowledge sharing patterns. Based on the knowledge sharing domain model, knowledge sharing and its context cannot be separated. With the knowledge sharing domain model we can also show that a pattern approach is good for this purpose. Compared to many other pattern formats, the knowledge sharing pattern format selected for the concept also has some knowledge sharing related parts and especially parts that combine the knowledge sharing with the environment of knowledge sharing.

The fifth sub-question for supporting was: *How can knowledge-sharing situations in software engineering be supported?* The answer and the contribution is the process for creating and evaluating an organizational pattern language, and the Knowledge Sharing Pattern Language. These are the main results from the supporting part. The process of creating and evaluating an organizational pattern language is summarized in Chapter 7.2. It is a process that can be reused for some other pattern language. It has not yet been tested to create any other organizational pattern domain, but it is quite general and has considerable potential for reuse.

Knowledge Sharing Pattern Language includes several knowledge sharing patterns every one of which provides support for different knowledge sharing situations. One of the purposes of the Knowledge Sharing Pattern Language is to share information about what the required knowledge sharing would mean in practice and about the various ways of sharing knowledge.

The sixth sub-question was: *How can we assure sufficiently wide coverage of knowledge-sharing solutions?* The answer for this is the Knowledge Sharing Goal Tree. Knowledge Sharing Pattern Language needs to have wide enough coverage to include the richness of knowledge sharing in software engineering. This is assured through creating first the Knowledge Sharing Goal Tree, and taking care that the Knowledge Sharing Pattern Language has patterns for each leaf of the Knowledge Sharing Goal Tree. Also the evaluation of the applicability of the Knowledge Sharing Pattern Language utilizing the scenario-based approach provided some additional confidence in the coverage of the knowledge-sharing solutions.

### 12.2.3 Contributions – Evaluating

Evaluation of an organizational pattern language is not trivial. The patterns, before capturing as pattern descriptions, exist in practice and evidence of their success is gained through applying the patterns several times in different environments. This, among other things, gives challenges to the evaluation. One of the contributions of this study is the evaluation technique for an organizational pattern language. Those are summarized in Section 10.5 and in Chapter 10 discussed and used in the context of evaluating the Knowledge Sharing Pattern Language.

## 12.3 Reviewing the Research Method

This research followes the design science research method. It is based on an iterative cycle of developing/building and justifying/evaluating artifacts. Hevner et al. (2004) have introduced seven guidelines for design science research (Table 27). The following is descriptive of how they apply to this study.

The first guideline is *Design as an Artifact*, meaning that design science research must provide an artifact. Hevner et al. (2004) define four types of IT artifacts. Those are: constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices), and instantiations (implemented and prototype systems).

The contributions of this research are introduced in Section 1.4 and discussed in Section 12.2. The resulting main artifacts are the Knowledge Sharing Framework (KSF) and the Knowledge Sharing Pattern Language. The first one is the main result from the understanding part and the last one from the supporting part of this research. KSF could be categorized as a model. The single patterns of the Knowledge Sharing Pattern Language are methods, but together those form an instantiation, a prototype system of knowledge sharing patterns supporting knowledge sharing in software engineering.

The second guideline is *Problem Relevance*, highlighting the need of solving relevant, real life (business) problems with the artifacts. In this research the research question is based on practical problems in software engineering work having connections to difficulties in or lack of knowledge sharing. The need to improve software engineering work performance through supporting knowledge sharing rose based on practical problems.

Table 27. Design Science Research Guidelines (Hevner et al. 2004).

| **Guideline** | **Description** |
|---|---|
| Guideline 1: Design as an Artifact | Design-science research must provide a viable artifact in the form of a construct, a model, a method or an instantiation. |
| Guideline 2: Problem Relevance | The objective of design-science research is to develop technology-based solutions to important and relevant business problems. |
| Guideline 3: Design Evaluation | The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. |
| Guideline 4: Research Contributions | Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. |
| Guideline 5: Research Rigor | Design-science research relies upon the application of rigorous methods in both, the construction and evaluation of the design artifact. |
| Guideline 6: Design as a Search Process | The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment. |
| Guideline 7: Communication of Research | Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences. |

The third guideline, *Design Evaluation* is a reminder that evaluation is a crucial component of the research. Since design is an iterative process, the evaluation needs to provide feedback to the development. The actual evaluation of the main artifacts of this research is introduced in Part IV of this thesis. KSF was developed through a conceptual study based on the foundations defined earlier in this thesis. The result is evaluated utilizing the case study method. This development-evaluation process is iterative in such a way that the case study results and feedback affected the finalization of the KSF.

The evaluation of the Knowledge Sharing Pattern Language included three different perspectives. First, the single patterns were validated, then the coverage of the Knowledge Sharing Pattern Language was ensured, and finally, the usability of the pattern language was evaluated.

The fourth guideline, *Research Contributions* states that design science research must provide clear contributions. It has the potential to produce three types of contributions: the design artifact, foundations, and methodologies. The main contributions are introduced in Section 1.4 and discussed in Section 12.2. The KSF model and the Knowledge Sharing Pattern Language are examples of design artifacts. At the same time those are clearly contributions to the foundations of the scientific knowledge base. The development of the Knowledge Sharing Pattern Language was implemented through different phases which are introduced in this thesis and summarized in Chapter 7.2. The creation of it included several intermediate results, which are also contributions of this research. The creation of the Knowledge Sharing Pattern Language utilized the knowledge foundations and the results then add new knowledge to the foundations. The contributions from this study to scientific methodologies include the process for creating an organizational pattern language (Chapter 7.2), and techniques for evaluating an organizational pattern language (Section 10.5).

The fifth guideline, *Research Rigor*, is a reminder that design science research requires the application of rigorous methods in both the construction and evaluation of the designed artifact. KSF was created through conceptual study based on the knowledge foundations, and finally evaluated through case study, and through applying it systematically to studying the knowledge sharing potential of some selected software development methodologies. Similarly, the Knowledge Sharing Pattern Language was developed through steps based on the KSF and knowledge foundations. The evaluation of the Knowledge Sharing Pattern Language is introduced in Chapter 10.

The sixth guideline, *Design as a Search Process*, is a reminder that design is essentially an iterative search process to discover an effective solution to a problem. The design of the Knowledge Sharing Pattern Language was an iterative process utilizing several improvement loops. The selection, to have a pattern-based approach for supporting knowledge sharing, however, did not include any comparisons to other possible approaches/tools. The selection was made purely based on having the understanding of patterns as a potential tool and then verifying the applicability with the knowledge sharing domain model.

The final, seventh guideline, *Communication of Research*, is a reminder that the results need to be introduced both to a technology-oriented and to a management-oriented audience. Both audiences have different perspectives to the artifact and need different information about it. To

cover these both audiences, the main results of this research werepublished as articles (see Section 1.4) and in the organization being studied.

The rigor and relevance of this research is ensured through compliance with a systematic research method and especially with the seven guidelines of design science research (Hevner et al. 2004).

## 12.4 Limitations of the Study

The limitations of this study are discussed here to provide a better understanding of the level of generalization that can be made based on the results. Three main limitations are: implementation only in one organization, missing practical experience of the use of the Knowledge Sharing Pattern Language, and the researcher being actor in the organization.

The empirical environment for this study was one unit of a global software engineering organization. For example, the case studies piloting the Knowledge Sharing Framework (KSF) were made there. The case studies also produced some raw material for the Knowledge Sharing Pattern Language. To overcome this limitation, the empirical material has not been the only raw material used. In addition to that, literature of widely recognized methods is utilized. Also, the structuring of the Knowledge Sharing Pattern Language was not done based on the results from the case study, but a separate Knowledge Sharing Goal Tree was made based on literature to structure the pattern language.

Knowledge Sharing Pattern Language has not yet been thoroughly used in practice. As discussed in Section 10.1, however, patterns, described earlier as a pattern description, have already had successful instances, based on which the pattern description was constructed. Then, the pattern evaluation is actually more like a justification that such instances have existed. After proper introduction of the Knowledge Sharing Pattern Language in practice, feedback can be gained to improve the patterns and to see which patterns are found to be more useful than others. The resulting knowledge sharing patterns have not been evaluated by the wider pattern community, but one successful evaluation workshop was arranged with people not having previous experience of the Knowledge Sharing Pattern Language.

Part of the time, this researcher has also been a managerial actor in the organization which was studied. Coghlan (2001) lists several possible problems that can arise from this type of setting. For example, the following issues (Coghlan 2001) could arise in this case:

1. While having insights and experience before the research program, the insider actor may assume too much and not probe as deeply as those ignorant of the situation might do.
2. The inside actor may find it difficult to obtain relevant data, because of the need to traverse departmental, functional or hierarchical boundaries or because as an insider she/he may be denied deeper access, which may not be denied an outsider.
3. Role duality: the inside actor may be at the same time in an organizational manager role and in a researcher role.

To avoid assuming too much, the golden skill of listening was used as much as possible, especially to insure that the researcher did not answer on behalf of other persons. Also the case study findings were looked through by other actors in the organization. In a lean organization,

such as the organization being studied, the departmental, functional and hierarchical boundaries are very scarce, and the researcher was not a supervisor for any persons interviewed in case studies. The position of the researcher did not have direct dependencies to any of the persons interviewed. In this organization, for the insider, it is much easier to go deeper than an outsider could do because of strict information security limitations towards outsiders.

The dual role was only a minor problem in this situation, because the researcher was a sort of internal consultant by virtue of her position. This and the climate that was created of continuous improvement, has helped in introducing development projects which these research projects actually are from the point of view of the organization.

# 13 CONCLUSIONS

Implementing this study was an interesting journey with many important lessons to be learned. Starting from the definition of the term itself,knowledge sharing is a challenging area to study. In this thesis, knowledge sharing is defined to be the process of perspective making and taking according to Boland and Tenkasi (1995). This process is always taking place between people, because knowledge is created individually.

Based on practice, knowledge sharing seems to play a central role in the problems appearing in software engineering work and projects, but it would be naïve to claim that all such problems could be avoided following the guidance from this study. Based on the results of this thesis, however, new possibilities are gained to identify several knowledge sharing problems early enough to react, or in some cases, to identify a risk and to mitigate it with proper actions. Key points here are the identification of possible knowledge sharing problems and pro-activity in avoiding such problems.

Understanding knowledge sharing only as time-and-place-bound activity would be a drastic restriction, effacing the richness of practical knowledge sharing. We need to accept that knowledge sharing could include transitions, for example, between knowledge and information. An example could be one person creating and documenting software design and, at another time and place, another person continuing and creating software code based on the design document. Based on her knowledge the first person makes design decisions and records the results as a design document. The design document is then information that is transferred to the second person. The second person reads the design document (information) and decides, based on the document and her earlier experience (knowledge), what the resulting source code should look like. Knowledge has been shared here, but it must be noticed that the created knowledge is not exactly the same knowledge that was originally shared. Prior knowledge of the receiver affects the knowledge creation in knowledge sharing.

Knowledge Sharing Framework (KSF) was defined in order to understand more completely the environment of knowledge sharing and to provide a tool to analyze the environment. KSF has three dimensions: knowledge sharing interfaces, software engineering activity types and project lifecycle phases. The selection of the knowledge sharing interfaces especially and the idea of separately studying knowledge sharing inside a project team within an organization and between organizations (special case used: customer-supplier-relationship) were instrumental in gaining new insights. Understanding the knowledge sharing interfaces inside an organization, in particular, was crucial. An example could be the appreciation of the importance of how to make the knowledge of experts in an organization available and not assigning the experts to only one project at a time. As a single finding, this is not new. The ability of KSF in helping to produce findings from several different perspectives was promising.

Some kind of tool was necessary in order to improve knowledge sharing. From the very beginning it was clear that a new software engineering process with improved knowledge sharing properties would not be reasonable and useful. Something else was needed to propose improvements to the rich area of software engineering work. The solution was to introduce

knowledge sharing patterns which could be used in parallel with already existing software engineering processes. As described in Chapter 8, the resulting knowledge sharing patterns can be used in different ways. Actually, some of the usages were not planned at all, but were identified later when utilizing the patterns. For example, the resulting pattern language, Knowledge Sharing Pattern Language, in addition to supporting single knowledge sharing situations, proved to be a good structure to describe and introduce knowledge sharing in a software engineering organization in general.

One by-product of this study is a process of creating an organizational pattern language. This process has not been properly evaluated here because it is not one of the main contributions. We have discussed its applicability briefly for other contexts in Section 7.2. This process should be possible and perhaps even reasonable to apply in other contexts. The main challenge in this process is how to validate the pattern language and the single patterns. The "true" validation comes only through applying the patterns and the language in practice. Here, we wanted to find ways to validate patterns and the language before extensive introduction in practice. If this kind of validation proves to be successful, it has the potential to facilitate the introduction of the patterns in practice. This means that less iteration would be needed after the introduction. As a result of the evaluation and the study on related work, several ideas for improving the pattern language emerged. Interfaces to other relevant patterns and pattern languages also need to be referenced from the Knowledge Sharing Pattern Language. Individual patterns in the language will presumably mature gradually as a result of practical application.

From a theoretical research perspective, more effort will be required to develop motivational support for knowledge sharing, i.e. reasons for sharing knowledge. This kind of research could result, for example, in improved forces descriptions in the knowledge sharing patterns. One design research topic might also be how to further improve the use of the Knowledge Sharing Pattern Language to be a knowledge-base type of a tool. Knowledge Sharing Pattern Language could be the basic structure in a knowledge base that has the possibility of easily adding lessons learned and yielding recommendations for single practices. The tool could automatically suggest relevant sequences of knowledge sharing patterns. It could also support the linking of patterns with an organization's existing processes.

To conclude, several results of this study contribute to software engineering by creating a better understanding of the importance of knowledge sharing and providing support for improving it in practice. This study has two major contributions. First, it is shown that the Knowledge Sharing Framework (KSF) helps to understand the current situation with knowledge sharing and to define targets in which a software engineering organization can improve knowledge sharing. A literature survey of related work indicates that KSF is a novel tool for studying knowledge sharing in organizations. Second, the Knowledge Sharing Pattern Language is developed to support knowledge sharing and also to support understanding of the importance of knowledge sharing in an organization. In this way, the thesis relies on one of the most successful and fundamental ideas in software engineering, the pattern concept.

# References

Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J. (2002). Agile software development methods: Review and analysis. VTT PUBLICATIONS 478, Espoo.

Ackoff, R.L. (1989). From Data to Wisdom. Journal of Applied Systems Analysis, 16, 3-9.

van Aken, J.E. (2005). Management Research as a Design Science: Articulating the Research Products of Mode 2 Knowledge Production in Management. British Journal of Management, 16(1), 19-36.

Alexander, C. (1999). The Origins of Pattern Theory: The Future of the Theory, and the Generation of a Living World. IEEE Software, 16(5), 71-82.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, M. and Angel, S. (1977). A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York.

Althoff, K-D. and Pfahl, D. (2003). Making Software Engineering Competence Development Sustained through Systematic Experience Management. In book Aurum et al. (2003), 269-294.

Andrade, J., Ares, J., Garcia, R., Pazos, J., Rodriguez, S., Rodriguez-Paton, A. and Silva, A. (2007). Towards a lessons learned system for critical software. Reliability Engineering & System Safety, 92(7), 902-913.

Ardichvili, A., Maurer, M., Li, W., Wentling, T. and Stuedemann, R. (2006). Cultural influences on knowledge sharing through online communities of practice. Journal of Knowledge Management, 10(1), 94-107.

Arogate, L. and Ingram, P. (2000). Knowledge Transfer: A Basis for Competitive Advantage in Firms. Organizational Behavior and Human Decision Processes. 82(1), 150-169.

Artto, K.A. (1998). A Management Framework for a Project Company. In proceedings of the 14th IPMA World Congress on Project Management, Ljubljana, Slovenia, 677-681.

Aurum, A. (2003). Supporting Structures for Managing Software Engineering Knowledge. In book Aurum et al. (2003), 69-72.

Aurum, A., Jeffery, R., Wohlin, C. and Handzic, M. (Eds.) (2003). Managing Software Engineering Knowledge. Springer-Verlag, Berlin, Heidelberg.

Baalen van, P., Bloemhof-Ruwaard, J. and Heck van, E. (2005). Knowledge Sharing in an Emerging Network of Practice: The Role of a Knowledge Portal. European Management Journal, 23(3), 300-314

Babar, M.A. and Gorton, I. (2007). Architecture Knowledge Management: Challenges, Approaches and Tools. 29th International Conference on Software Engineering (ICSE 2007), May 2007, 170-171.

Barnes, B.H. and Bollinger, T.B. (1991). Making Reuse Cost-Effective. IEEE Software, 8(1), 13-24.

Basili, V.R., Caldiera, G. and Rombach, H.D. (1994). Experience Factory. In Marciniak, J.J. (Ed) Encyclopedia of Software Engineering. John Wiley and Sons, UK, 469-476.

Beck, K. (2000). Extreme Programming Explained: Embrace Change. The XP Series, Addison-Wesley.

Bell, D. (2003). UML Basics: Part II: The Activity Diagram. Rational Edge. November, 2003, 4-19.
http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/sep03/TheRationalEdge_September2003.pdf (last visited: June 2009).

Bellini, E., Canfora, G., Cimitile, A., Garcia, F., Piattini, M. and Visaggio, C.A. (2005). The Impact of Educational Background on Design Knowledge Sharing During Pair Programming: An Empirical Study. Third Biennial Conference, WM 2005, Kaiserslautern, Germany, Lecture Notes in Computer Science 3782, 455-465.

Birk, A., Dingsoyr, T. and Stalhane, T. (2002) Postmortem: never leave a project without it. IEEE Software, 19(3), 43-45.

Blackler, F. (1995). Knowledge, Knowledge Work and Organizations: An Overview and Interpretation. Organization Studies, 16 (6), 1021-1046.

Boehm, B. (2006). Value-Based Software Engineering: Overview and Agenda. In book Biffl, S., Aurum, A., Boehm, B., Erdogmus, H. and Grünbacher, P. (Eds.) Value-Based Software Engineering. Springer, 3-14.

Boehm, B. and Turner, R. (2004). Balancing Agility and Discipline: A Guide for the Perplexed. Pearson Education Inc., Addison-Wesley.

Boland, R.J. and Tenkasi, R.V. (1995). Perspective Making and Perspective Taking in Communities of Knowing. Organization Science 6 (4), 350-372.

Booch, G. (2008). Architectural Organizational Patterns. IEEE Software 25(3), 18-19.

Bosch, J. (2000). Design and use of software architectures: Adopting and evolving a product-line approach. ACM Press, Addison-Wesley.

Bosch-Sijtsema, P.M. (2004). A knowledge transfer framework for virtual projects. Int. J. Networking and Virtual Organizations, (2)4, 298-311.

Brown, J.S. and Duguid, P. (2001). Knowledge and Organization: A Social-Practice Perspective. Organization Science, 12 (2), 198-213.

Buschmann, F., Henney, K. and Schmidt, D.G. (2007). Past, Present, and Future Trends in Software Patterns. IEEE Software, 24(4), 31-37.

Casey, V. and Richardson, I. (2006). Project Management within Virtual Software Teams. In Proceedings of ICGSE'06 International Conference on Global Software Engineering, Florianapolis, Brazil, October 2006. IEEE Computer Society.

Chau, T., Maurer, F. and Melnik, G. (2003). Knowledge Sharing: Agile Methods vs. Tayloristic Methods. Proceedings of the twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03).

Chau, T. and Maurer, F. (2004). Knowledge Sharing in Agile Software Teams. In proceedings of Symposium on Logic Versus Approximation, Kaiserslautern, Germany, October 2003.

Chrissis, M.B., Konrad, M. and Shrum, S. (2003). CMMI. Guidelines for Process Integration and Product Improvement. SEI Series in Software Engineering. Addison-Wesley.

Clements, P., Kazman, R. and Klein, M. (2002). Evaluating Software Architectures: Methods and Case Studies. SEI Series in Software Engineering, Addison-Wesley.

Clements, P. and Northrop, L. (2002). Software Product Lines: Practices and Patterns. The SEI Series in Software Engineering. Addison-Wesley.

Coghlan, D. (2001). Insider Action Research Projects. Implications for Practicing Managers. Management Learning, 32(1), 49-60.

Cohen, M.D. and Sproull, L.S. (1996). Organizational Learning. SAGE Publications.

Cook, S.D.N. and Brown, J.S. (1999). Bridging Epistemologies: The Generative Dance Between Organizational Knowledge and Organizational Knowing, Organization Science 10(4), 381-400.

Coplien, J.O. (1995). A Development Process Generative Pattern Language. http://users.rcn.com/jcoplien/Patterns/Process/index.html (last visited: January 2009).

Coplien, J.O. (1996). Software Patterns. SIGS Books, New York.

Coplien, J.O. (1998). A Generative Development-Process Pattern Language. In book Rising, L. (Ed.) The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press, 243-300

Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall.

Covey, S.R. (1989). The 7 Habits of Highly Effective People: Powerfull Lessons in Personal Change. Free Press.

Damian, D. (2007). Stakeholders in Global Requirements Engineering: Lessons Learned from Practice. IEEE Software, 24(2), 21-27.

Davenport, T.H. and Prusak, L. (1998). Working Knowledge: How Organizations Manage What They Know. Harvard Business School Press, Boston, Massachusetts.

Desouza, K.C. (2003). Barriers to Effective Use of Knowledge Management Systems in Software Engineering. Communications of the ACM, 46(1), 99-101.

Dingsøyr, T., Bjørnson, F.O. and Shull, F. (2009). What Do We Know about Knowledge Management? IEEE Software, 26(3), 100-103.

Dingsøyr, T. and Conradi, R. (2003). Usage of Intranet Tools for Knowledge Management in a Medium-Sized Software Consulting Company. In book Aurum et al. (2003), 49-68.

Dixon, N.M. (2000). Common Knowledge: How Companies Thrive by Sharing What They Know. Harvard Business School Press, Boston, Massachusetts.

Dutoit, A.H. and Paech, B. (2003). Eliciting and Maintaining Knowledge for Requirements Evolution. In book Aurum et al. (2003), 135-155.

Dybå, T. (2003). A Dynamic Model of Software Engineering Knowledge Creation. In book Aurum et al. (2003), 95-117.

Earl, M. (2001). Knowledge Management Strategies: Toward a Taxonomy. Journal of Management Information Systems, 18(1), 215-233.

Ebert, C. De Man, J. and Schelenz, F. (2003). e-R&D: Effectively Managing and Using R&D Knowledge. In book Aurum et al. (2003), 340-359.

Edwards, J.S. (2003). Managing Software Engineers and Their Knowledge. In book Aurum et al. (2003), 5-27.

Ehrlich, K. (2003). Locating Expertise: Design Issues for an Expertise Locator System. In book Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise: Beyond Knowledge Management. MIT Press, 137-158.

Eisenhardt, K.M. (1989). Building Theories from Case Study Research. Academy of Management Review, (14(4), 532-550.

Espinosa, J.A., Slaughter, S.A., Kraut, R.E. and Herbsleb, J.D. (2007). Team Knowledge and Coordination in Geographically Distributed Software Development. Journal of Management Information Systems, 24(1), 135-169.

Farenhorst, R. (2006). Tailoring knowledge sharing to the architecting process. ACM SIGSOFT Software Engineering Notes. 31(5). SHaring and Reusing architectural Knowledge (SHARK'2006), article no. 3.

Farenhorst, R., Lago, P. and van Vliet, H. (2007). Prerequisities for Successful Architectural Knowledge Sharing. Proceedings of the 2007 Australian Softare Engineering Conference (ASWEC'07).

Fitzpatrick, G. (2003). Emergent Expertise Sharing in a New Community. In book Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise: Beyond Knowledge Management. MIT Press, 81-110.

Freeman, R.E. (1984). Strategic Management: A Stakeholder Approach. Boston, MA: Harper-Collins.

Fowler, M. (2004). UML Distilled: A Brief Guide to the Standard Object Modeling Language. Object Technology Series, Addison-Wesley.

Galin, D. (2004). Software Quality Assurance: From theory to implementation. Pearson Education, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series.

Gareis, R. (1996). The Application of the "New Management Paradigm" in the Project-Oriented Company. In Proceedings of IPMA´96 World Congress on Project Management, Paris, France, 687-689.

Haas, R., Aulbur, W. and Thakar, S. (2003). Enabling Communities of Practice at EADS. In book Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise: Beyond Knowledge Management. MIT Press, 179-198.

Hafiz, M., Adamczyk, P. and Johnson, R.E. (2007). Organizing Security Patterns. IEEE Software, 24(4), 52-60.

Handzic, M. (2003). Why Is It Important to Manage Knowledge? In book Aurum et al. (2003), 1-4.

Harrington, J. (1991). Business Process Improvement – the Breakthrough Strategy for Total Quality, Productivity and Competitiveness. McGraw-Hill Inc.

Hevner, A. R., March, S. T., Park, J. and Ram, S. (2004). Design Science in Information Systems Research. MIS Quarterly, 28(1), 75-105.

Hinds, P.J. and Pfeffer, J. (2003). Why Organizations Don't "Know What They Know": Cognitive and Motivational Factors Affecting the Transfer of Expertise. In book: Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise: Beyond Knowledge Management. The MIT Press, 3-26.

Humphrey, W.S. (1998). Foreword for the book: Zahran, S. Software Process Improvement: Practical Guidelines for Business Success. Addison-Wesley, xiii-xiv.

Huysman, M., and de Wit, D. (2003). A Critical Evaluation of Knowledge Management Practices. In book Ackerman, M., Pipek, V. and Wulf, V. Sharing Expertise: Beyond Knowledge Managemeng. The MIT Press, 27-55.

IEEE (2002). A special section on knowledge management in software engineering (several articles). IEEE Software, 19 (3), 14-15, 26-62.

ISO (2001). Software Engineering – Product Quality – Part 1: Quality Model. ISO/IEC 9126-1:2001.

Jaaksi, A., Aalto, J-M., Aalto, A. and Vättö, K. (1999). Tried & True Object Development: Industry-Proven Approaches with UML. Cambridge University Press.

Jacobson, I., Griss, M. and Jonsson, P. (1997). Software Reuse: Architecture, Process and Organization for Business Success. ACM Press, Addison-Wesley.

Jalote, P. (2003). Knowledge Infrastructure for Project Management. In book Aurum et al. (2003), 361-375.

Johnson, J. (2006). My Life is Failure: 100 Things You Should Know to be a Successful Project Leader. The Standish Group International, Inc. West Yarmouth, MA.

Jørgensen, M. (2007). Forecasting of software development work effort: Evidence on expert judgement and formal models. International Journal of Forecasting 23, 449-462.

Kanawattanachai, P. and Yoo, Y. (2007). The Impact of Knowledge Coordination on Virtual Team Performance over Time. MIS Quarterly, 31(4), 783-808.

Karni, R. and Kaner, M. (2005). Agile Knowledge-Based Decision Making with Application to Project Management. Third Biennial Conference, WM 2005, Kaiserslautern, Germany, Lecture Notes in Computer Science 3782, 400-408.

Kirchner, M. and Völter, M. (2007). Guest Editors' Introduction: Software Patterns. IEEE Software, 24(4), 28-30.

Komi-Sirviö, S., Mäntyniemi, A. and Seppänen, V. (2002). Toward a practical solution for capturing knowledge for software projects.

Koskinen, J., Salminen A. and Paakki, J. (2004). Hypertext support for the information needs of software maintainers. Journal of Software Maintenance and Evolution: Research and Practice, 16(3), 187-215.

Krogh von, G., Roos, J. and Slocum, K. (1994). An Essay on Corporate Epistemology. Strategic Management Journal, 15 (Special Issue: Strategy: Search for New Paradigms), 53-71.

Kylmäkoski, R. (2006). RaPiD7: A Collaborative Method for the Planning Activities in Software Engineering – Industrial Experiments. PhD thesis. Tampere University of Technology, Publication 582, Tampere, Finland.

Liebowitz, J. (2002). A look at NASA Goddard Space Flight Center's knowledge management initiative. IEEE Software, 19(3), 40-42.

Lindvall, M. and Rus, I. (2003). Knowledge Management for Software Organizations. In book Aurum et al. (2003), 73-94.

Lindvall, M., Rus, I. and Sinha, S.S. (2003). Software systems support for knowledge management. Journal of Knowledge Management, 7(5), 137-150.

Lowe, D. (2003). Emergent Knowledge in Web Development. In book Aurum et al. (2003), 157-175.

Manns, M.L. and Rising, L. (2005). Fearless Change: Patterns for Introducing New Ideas. Pearson Education Inc., Addison-Wesley.

May, D. and Taylor, P. (2003). Knowledge Management with Patterns. Communications of the ACM, 46(7), 94-99.

McCarthy, J. and McCarthy, M. (2006). Dynamics of Software Development. Microsoft Press, Redmond, Washington.

Mestad, A., Myrdal, R., Dingsoyr, T. and Dybå, T. (2007). Building a learning Organization: Three Phases of Communities of Practice in a Software Consulting Company. Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07), IEEE.

Mili, H., Mili, F., and Mili A. (1995). Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering, 21(6), 528-562.

Nakakoji, K. (2006). Supporting Software Development as Collective Creative Knowledge Work. Proceedings of the Second International Workshop on Supporting Knowledge Collaboration in Software Development (KCSE2006), Ye, Y. and Ohira, M. (Eds.), NII, Tokyo, September, 2006.

Nonaka, I. (1994), A Dynamic Theory of Organizational Knowledge Creation. Organization Science, 5 (1), 14-37.

Nonaka, I., von Krogh, G. and Voelpel, S. (2006). Organizational Knowledge Creation Theory: Evolutionary Paths and Future Advances. Organization Studies, 27(8), 1179-1208.

Nonaka, I. and Takeuchi, H. (1995). The Knowledge Creating Company. Oxford University Press.

Oliver, G.R., D'Ambra, J. and Van Toorn, C. (2003). Evaluating an Approach to Sharing Software Engineering Knowledge to Facilitate Learning. In book Aurum et al. (2003), 119-134.

OMG (2007). OMG Unified Modelling Language (OMG UML), Superstructure, V2.1.2. http://www.omg.org/docs/formal/07-11-02.pdf (last visited: June 2009).

Orlikowski, W.J. (2002). Knowing in Practice: Enacting a Collective Capability in Distributed Organizing. Organization Science, 13(3), 249-273.

Paasivaara, M. (2005). Communication Practices in Inter-Organizational Product Development. Dissertation for the degree of Doctor of Science in Technology. University of Technology, Espoo, Finland.

Peine, H. (2005). Rules of thumb for secure software engineering. Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA.

Pender, T. (2003). UML Bible. Wiley Publishing Inc. Indianapolis, IN, USA

Persson, A., Stirna, J., Dulle, H., Hatzenbichler, G. and Strutz, G. (2003). Introducing a Pattern Based Knowledge Management Approach: The Verbundplan Case. Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA'03).

Peresson, A. and Stirna, J. (2006). How to Transfer a Knowledge Management Approach to an Organization – A Set of Patterns and Anti-patterns. In proceedings Reimer, U. and Karagiannis, D. (Eds.): PAKM 2006, LNAI 4333, 243-252.

Petter, S., Mathiassen, L. and Vaishnavi, V. (2007). Five Keys to Project Knowledge Sharing. IT Professional, 9(3), 42-46.

Polanyi, M. (1966). The Tacit Dimension. Routledge & Kegan Paul, London.

Ramasubramanian, S. and Jagadeesan, G. (2002). Knowledge management at Infosys. IEEE Software, 19(3), 53-55.

Ramesh, B. (2002). Process knowledge management with traceability. IEEE Software, 19(3), 50-52.

Reifer, D.J. (2002). A Little Bit of Knowledge is a Dangerous Thing. IEEE Software, 19 (3), 14-15.

Rising, L. (2007). Understanding the Power of Abstraction in Patterns. IEEE Software 24(4), 46-51.

Rosenberg, L.H. (2003). Lessons Learned in Software Quality Assurance. In book Aurum et al. (2003), 251-268.

Royce, W.W. (1970). Managing the Development of Large Software Systems. In proceedings of IEEE Wescon, 1-9.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991). Object-Oriented Modeling and Design. Prentice Hall.

Rus, I. and Lindvall, M. (2002). Knowledge Management in Software Engineering. IEEE Software, 19(3), 26-38.

Schalken, J., Brinkkemper, S. and van Vliet, H. (2006).A Method to Draw Lessons from Project Postmortem Databases. Software Process: Improvement and Practice, 11(1), 35-46.

Schneider, K, von Hunnius, J-P. and Basili, V.R. (2002). Experience in implementing a learning software organization. IEEE Software, 19(3), 46-49.

Schuler, D. (2008). Liberating Voices: A Pattern Language for Communication Revolution. The MIT Press, Cambridge, Massachusetts, London, England. Patterns available also in Internet: http://www.publicsphereproject.org/patterns/pattern-table-of-contents.php (last visited: January 2009).

Shepperd, M. (2003). Case-Based Reasoning and Software Engineering. In book Aurum et al. (2003), 181-198.

van Solingen, R. (2003). In-Project Learning by Goal-oriented Measurement. In book Aurum et al. (2003), 319-337.

van Solingen, R., Kusters, R. and Trienekens, J. (2003). Practical Guidelines for Learning-Based Software Product Development. In book Aurum et al. (2003), 299-317.

Sommerville, I. (2006). Software Engineering. 8th Edition. Pearson Education.

Sommerville, I. and Sawyer, P. (1997). Requirements Engineering: A Good Practice Guide. John Wiley & Sons Ltd, Chichester, UK.

Strohmaier, M. and Lindstaedt, S.N. (2005). Application of Knowledge Problem Patterns in Process Oriented Organizations. In Proceedings of WM 2005, LNAI 3782, 259-268.

Sugumaran, V., Park S. and Kang, K. (2006). Software Product Line Engineering. Communications of the ACM, 49(12), 29-32.

Sutcliffe, A. (2002). The Domain Theory: Patterns for Knowledge and Software Reuse. Taylor & Francis, Inc.

Sveiby, K.E. and Lloyd, T. (1987). Managing Knowhow: Add value by valuing creativity. Bloomsbury, London.

Star, S.L. and Griesemer, J.R. (1989). Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkley's Museum of Vertebrate Zoology, 1907-39. Social Studies of Science, 19, 387-420.

Tiwana, A. and Bush, A. A. (2005). Continuance in Expertise-Sharing Networks: A Social Perspective. IEEE Transactions on Engineering Management. 52(1), 85-101.

Tiwana, A. and Ramesh, B. (2001). Integrating Knowledge on the Web. IEEE Internet Computing, 5 (3), 32-39.

Turner, K.L. and Makhija, M.V. (2006). The Role of Organizational Controls in Managing Knowledge. Academy of Management Review, 31 (1), 197-217.

Vegas, S., Juristo, N. and Basili, V.R. (2003). A Process for Identifying Relevant Information for a Repository: A Case Study for Testing Techniques. In book Aurum et al. (2003), 199-230.

Verner, J.M. and Evanco, W.M. (2003). An Investigation into Software Development Process Knowledge. In book Aurum et al. (2003), 29-47.

Vesiluoma, S. (2005). Mining Knowledge Sharing Patterns. In proceedings of 28th Information Systems Research Seminar in Scandinavia (IRIS 28), Kristiansand, Norway.

Vesiluoma, S. (2006). Improving Knowledge Sharing in Software Engineering. International Transactions on Systems Science and Applications. 1(2):167-173. The same article has been introduced also in Evaluation of Novel Approaches to Software Engineering (ENASE), Erfurt, Germany.

Vesiluoma, S. (2007a). Making Knowledge Sharing Visible in Software Engineering. In proceedings of ECIS 2007, 15th European Conference on Information Systems, St. Gallen, Switzerland, 540-551.

Vesiluoma, S. (2007b). Knowledge Sharing Pattern Language. In Berki, E., Nummenmaa, J., Sunley, I., Ross, M. and Staples, G. (Eds.) Software Quality in the Knowledge Society. The

British Computer Society, 257-271. The same article has been introduced also in Software Quality Management conference, SQM 2007, Tampere, Finland.

Vesiluoma, S. (2007c). Software Development Methods and Knowledge Sharing. In Tiainen, T., Isomäki, H., Korpela, M., Mursu, A., Paakki, M-K., and Pekkola, S. (Eds.) Proceedings of 30th Information Systems Research Seminar in Scandinavia, IRIS30 Tampere, Finland, Department of Computer Sciences, University of Tampere, Finland, Series of Publications: D - Net Publications D-2007-9, September 2007, 262-278.

Vlissides, J. (1998). Pattern Hatching: Design Patterns Applied. Addison-Wesley software pattern series.

Ward, J. and Aurum. A. (2004). Knowledge Management in Software Engineering – Describing the Process.

Wareham, J. and Gerrits, H. (1999). De-Contextualizing Competence: Can Business Best Practice be Bundled and Sold? European Management Journal, 17(1), 39-49.

Wasserman, S. and Faust, K. (1994). Social Network Analysis: Methods and Applications. Cambridge University Press.

Wei, C-P., Hu, J-H. and, Chen H-H. (2002). Design and evaluation of a knowledge management system. IEEE Software, 19(3), 56-59.

Weiss, D.M. and Lai, C.T.R. (1999). Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley.

Welicki, L., Lovelle, J.M.C. and Aquilar, L.J. (2006). Patterns Meta-Specification and Cataloging: Towards Knowledge Management in Software Engineering. In proceedings of OOPSLA'06, Portland, Oregon, USA, October 2006, 679-680.

Winn, T. and Calder, P. (2002). Is This a Pattern? IEEE Software, 19(1), 59-66.

Välimäki, A., Vesiluoma, S. and Koskimies, K. (2008). Scenario-Based Assessment of Process Pattern Languages. In Bomarius, F. (Ed.). Proceedings of Product-Focused Software Process Improvement: 10th International Conference. PROFES 2009, Oulu, Finland LNBIB 32, Springer-Verlag, Berlin Heidelberg, 245-259.

Zahran, S. (1998). Software Process Improvement: Practical Guidelines for Business Success. SEI Series in Software Engineering, Addison-Wesley.

Zelany, M. (1987). Management Support Systems: Towards Integrated Knowledge Management. Human Systems Management, 7(1), 59-70.

Zhuge, H. (2006). Knowledge flow network planning and simulation. Decision Support Systems, 42(2), 571-592.

# Appendixes

# 14 CASE 'UNDERSTANDING' INTERVIEW QUESTIONS

After the basic data, the original Finnish questions first and then the translation.

**Basic data regarding the person**

1 Name
2 Duration while in the project (what is it like compared to the project duration?)
3 Role in the project
4 Office(s) where located and where the project was implemented

**Henkilön oma arvio projektista - Person's own evaluation regarding to the project**

5 Miten kuvailisit projektia? – How would you describe the project?
6 Mitkä asiat projetissa ovat mennet hyvin? – What things have gone well in the project?
7 Missä asioissa on ollut ongelmaa / olisi kehitettävää? – Where have the problems been? / What would need improvement?
8 Arvioi projektin toteutusta tavoitteeseen verrattuna asteikolla 1-5 (5= erinomainen) – Evaluate the project implementation compared to the objectives using scale 1-5 (5 = excellent).
9 Miten tyytyväinen olet omaan toimintaasi projektissa samalla asteikolla? – How satisfied are you with your own activities in the project, use the same scale?
10 Keiden kanssa olet eniten ollut yhteistyössä projektin aikana, missä asioissa? - With whom have you cooperated most during the project? In which matters?

**Projektin elinkaari – Project life cycle**

<u>Myyntivaihe - Sales phase</u>

11 Mitä näkyvyyttä sinulla oli myynnin aikaiseen suunnitteluun? – What visibility did you have to the planning/estimating during the sales phase?
12 Millainen pohja myynnin aikana valmistellut suunnitelmat olivat projektille? – What kind of basis did the plans/estimates, made during the sales, give to the project?
13 Oliko projektin siirto myynnistä projektitoteutukseen onnistunut? Saiko projektipäällikkö riittävästi tietoa projektin toteutuksen aloittamiseksi? – Was the transfer from sales to project implementation successful? Did the project manager get enough knowledge to start the project?

<u>Projektin aloittaminen – Project establishment</u>

14 Miten sinut perehdytettiin projektiin? – How the project was introduced to you?
15 Saitko helposti kokonaiskuvan (millainen projekti, sen tavoitteet jne.)? – Did you easily get the general view of the project (what kind of a project, it's objectives etc.)?
16 Saitko helposti kuvan mitä sinulta odotettiin ja miten se liittyi kokonaisuuteen? – Did you easily get the understanding of what was expected from you and how did it relate to the project entity?

Projektin aikana – Project implementation

17 Millaisia muutoksia projektin aikana tapahtui? – What kind of changes happened during the project?
18 Miten muutokset hallittiin? – How changes were managed?
19 Kulkiko tieto muutoksista niille joiden työtä ne koskivat? – Did the information/knowledge about the changes reach the persons affected by those changes?
20 Miten tulosten oikeellisuus varmistettiin projektin aikana? – How were the results of the project verified/validated during the project?
21 Olitko yhteydessä asiakkaaseen, missä asioissa? – Did you have direct contact with the customer, in what matters?
22 Miten asiakasyhteydenpito toimi? – How did communication with the customer work?

Projektin päättäminen – Project closure

23 Oliko projektissa helppo saada asiakkaan hyväksyntä? – Was it easy to get the acceptance from the customer?
24 Saatiinko/pyydettiinkö hyväksyntää asiakkaalta monta kertaa projektin aikana vai vasta projektin lopussa? – Was acceptance received several times during the project or only at the end of the project?
25 Mitä projektista opittiin? – What lessons were learned from the project?

**Tiedon/tietämyksen jakaminen projektitiimissä – Knowledge sharing in the project team**

26 Mistä asioista keskustelitte projektitiimissä projektin aikana? – What matters were discussed in project team during the project?
27 Mitä tietoa/tietämystä jaettiin projektitiimissä? – What knowledge was shared in the project team?
28 Mitä tietoa/tietämystä olisi pitänyt jakaa projektitiimissä? – What knowledge should have been shared in the project team?

29 Oliko tiimissänne etäjäseniä? Miten tiedon/tietämyksen vaihto sujui heidän kanssaan (mitä tietoa/tietämystä, miten kommunikoituna)? – Did your team have distant members? How the knowledge sharing was implemented with them (what knowledge, how communicated)?

**Tiedon/tietämyksen jakaminen organisaatiotasolla – Knowledge sharing at the organizational level**

30 Pystyttiinkö aikaisempien projektien tuotoksia hyödyntämään tässä projektissa? – Could the results of earlier projects be utilized in this project?

31 Miten varmistettiin, että ne olivat sopivia tälle projektille? – How was it assured that the results were appropriate for this project?

32 Mitä tämän projektin tuotoksia voitaisiin hyödyntää seuraavissa projekteissa? – What results from this project could be utilized in subsequent projects?

33 Miten resurssien hallinta onnistui tässä projektissa? Oliko kaikilla projektiin varatuilla henkilöillä sopiva työmäärä? – How resource management succeeded in this project? Did the people booked to the project have a suitable amount of work during the project?

34 Miten resurssimäärien muutokset sujuivat projektin aikana? Miten resurssineuvottelut hoidettiin? – How the changes in the amounts of resources were handled during the project? How these changes were negotiated?

**Tuleeko mieleen jotain muuta? – Anything else**

35 Muuta? – Anything else?

36 Terveiset? – Greetings?

# 15 KNOWLEDGE SHARING PATTERNS

This section includes the knowledge sharing patterns of the Knowledge Sharing Pattern Language. The pattern language is in practice created as interlinked web pages. Here, the links are marked, but not linked to each other.

In addition to the actual pattern descriptions, first the knowledge sharing pattern Catalog is introduced including a list of all knowledge sharing patterns in the Knowledge Sharing Pattern Language. Then the root pattern KSP00 has been introduced and after that all knowledge sharing patterns in the order of their identifier.

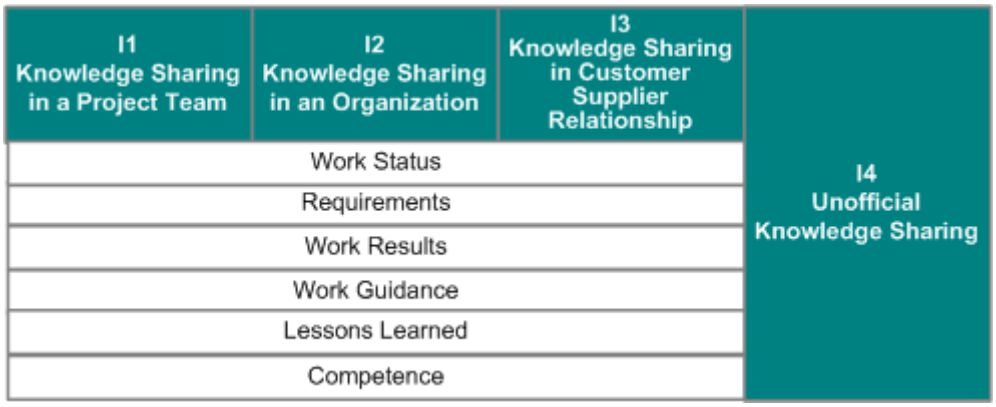
## 15.1 Knowledge Sharing Pattern Catalog

The following table includes the patterns introduced on this web-site for Knowledge Sharing Pattern Language. The patterns are listed in the order that they have been created.

| Id | Name | First Created | I1 | I2 | I3 | I4 | TK |
|---|---|---|---|---|---|---|---|
| KSP00 | Improved Knowledge Sharing (root pattern / home) | 05.03.2004 | X | X | X | X | all |
| KSP01 | Project Support | 05.03.2004 | | X | | | Comp |
| KSP02 | Named Experts | 05.03.2004 | | X | | | Comp |
| KSP03 | Assigned Experts | 05.03.2004 | | X | | | Comp |
| KSP04 | Trust or Check | 17.04.2006 | X | | | | WS |
| KSP05 | Shared Understanding | 08.09.2006 | | | X | | R |
| KSP06 | Discovered Bones | 13.09.2006 | | | X | | R |
| KSP07 | Reference Requirements | 11.10.2006 | | X | | | R |
| KSP08 | Created Skeleton | 11.10.2006 | X | | | | R |
| KSP09 | Schedule Baseline | 17.10.2006 | X | | | | WS |
| KSP10 | Known Status of Projects | 17.10.2006 | | X | | | WS |
| KSP11 | Informed Customer | 17.10.2006 | | | X | | WS |
| KSP12 | Managed Versions | 18.10.2006 | X | | | | WR |
| KSP13 | Quickly Made | 18.10.2006 | | X | | | WR |
| KSP14 | Release | 18.10.2006 | | | X | | WR |
| KSP15 | Discovered Lessons | 21.10.2006 | X | | | | LL |
| KSP16 | Established Experience Base | 21.10.2006 | | X | | | LL |
| KSP17 | Satisfied Customer | 21.10.2006 | | | X | | LL |
| KSP18 | Improved Competences | 22.10.2006 | X | | | | C |
| KSP19 | Assured Resources | 22.10.2006 | | | X | | C |
| KSP20 | Initiated Communication | 15.01.2007 | | | | X | - |
| KSP21 | My Network | 15.01.2007 | | | | X | - |
| KSP22 | Work Guidance | 04.02.2007 | X | X | X | | WG |
| KSP23 | Not Wasted | 06.04.2007 | | X | | | R |
| KSP24 | Flexible Skeleton | 06.04.2007 | X | | | | R |
| KSP25 | Followed Progress | 07.04.2007 | X | | | | WS |
| KSP26 | Reuse Approach | 07.04.2007 | | X | | | WR |
| KSP27 | Contributed Experience Base | 07.04.2007 | | X | | | LL |
| KSP28 | Utilized Experience Base | 07.04.2007 | | X | | | LL |
| Id | Name | First Created | I1 | I2 | I3 | I4 | TK |

# 15.2 KSP00 Improved Knowledge Sharing in Software Engineering

(Root pattern for the Knowledge Sharing Pattern Language)




**Problem** An organization wanting to make work more effective through improved knowledge sharing.

**Initial Context** An organization in software project business.

**Roles** All roles in the organization.

**Forces** Software engineering uses knowledge as the raw material and requires several people to work together. New knowledge is gained at an individual level and to succeed an organization must have ways to utilize and share this knowledge as effectively as possible.

A project team has several connections outside the project team and also the individuals have much knowledge that could benefit the other project team members. Very often, however, people ether do not understand the need for knowledge sharing or they do not have a willingness for it.

A Software engineering organization in the project business consists of a base organization and of projects in different phases in their lifecycles. In addition to having proper knowledge sharing in single projects the knowledge sharing between projects and between projects and the base organization must work well, to have both successful project implementations and an organization achieving its objectives.

To execute a successful customer project the customer’s requirements must be clearly understood, communicated and further developed to requirements for the project. Fluent and well established customer communications are critical for the project success which includes getting continuous feedback from the customer.

Unofficial knowledge sharing is a powerful "tool" for achieving the goals of an organization. Unofficial knowledge sharing, however, is fully based on people voluntarily implementing it. It can not be directed very much. The direction comes mostly from restrictions (e.g. confidentiality) and support to right kind of motivation of

people. In the worst case, unofficial knowledge sharing can share messages/knowledge that is defective to an organization.

**Solution**

Knowledge sharing and better knowledge utilization must be implemented at several levels/ways in an organization. Knowledge sharing here is approached from two viewpoints: knowledge sharing interfaces and target knowledge streams.

To improve knowledge sharing establish/initiate:

| *Knowledge Sharing Interfaces* | *Target Knowledge Streams* |
|---|---|
| • I1 Knowledge Sharing in a Project Team<br>• I2 Knowledge Sharing in an Organization<br>• I3 Knowledge Sharing in Customer Supplier Relationship<br>• I4 Unofficial Knowledge Sharing | • Work Status<br>• Requirements<br>• Work Results<br>• Work Guidance<br>• Lessons Learned<br>• Competence |

**Resulting Context**

An organization having a culture of sharing knowledge and assuring that correct knowledge reaches the right people. Also, effective utilization of knowledge.

**Instances**

This pattern is utilized once to arrange knowledge sharing support in an organization and after that always when it needs to be redefined.

**Process Connection**

All processes of an organization, but especially project management and software development related processes.

## Background

This pattern is the main pattern for the whole knowledge sharing pattern language. The purpose for this pattern language has been to:

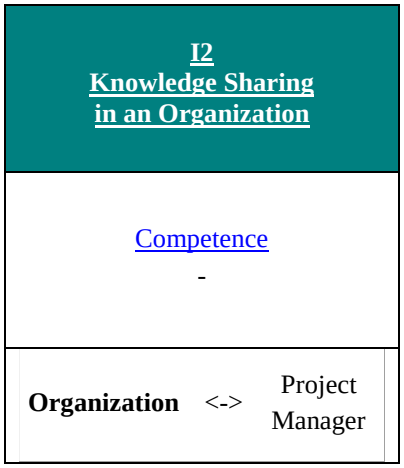
1. Make knowledge sharing visible in software engineering.
2. Create an understanding of the importance of knowledge sharing in software engineering.
3. To improve knowledge sharing in software engineering.

## See Also

- Knowledge Sharing Pattern Catalog
- Knowledge Sharing Pattern Format

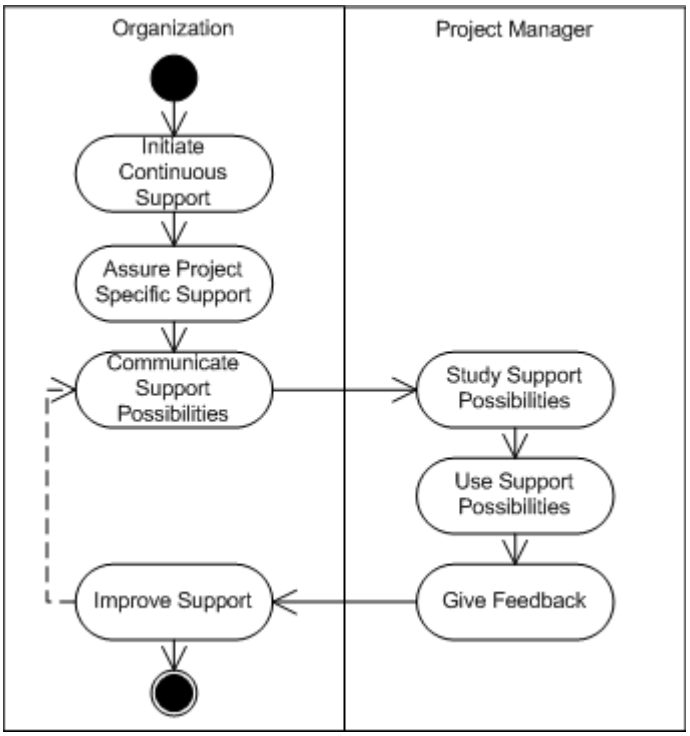
## 15.3 KSP01 Project Support

Dimensions and Knowledge Flow:




| | |
|---|---|
| **Problem** | A project manager is often very alone in a decision making situation and has to make a decision without adequate knowledge. The organization might have required knowledge elsewhere, but does not know how to make it available for projects in need. |
| **Initial Context** | Knowledge support for projects has not been systematically arranged at the organizational level. |
| **Roles** | *A Project Manager* and the *Organization* represented by a managerial actor responsible for multi-project management or manager of a project office function. |
| **Forces** | A project manager is very often alone in decision making situations. The decisions can not be made by others, but support can be given, for example in the form of discussions or expert advice. A project manager is in the middle of several knowledge flows and still can not have all the knowledge required in the project. (S)he needs to have persons with which to create the required understanding. |

**Solution**




Implement the following actions:

1. *Initiate Continuous Support* in the organization
   See: Named Experts, KSP02
2. *Assure Project Specific Support*. Look after that there are possibilities to arrange project specific support.
   (See: Assigned Experts, KSP03, but arrange this only after there is a specific project requiring this).
3. *Communicate (and study) support possibilities*. Make sure that Project Managers know how the competence support can be arranged (continuous and project specific) and how the support can be requested.
4. *Use support possibilities*, utilize those in a project.
5. *Give feedback* and *improve* available *support* based on the feedback.

The dash line from *Improve Support to Communicate Support Possibilties* is added to show that in practice the phases 3-5 are a continuous action.

**Resulting Context** Project teams and especially Project Managers in the organization feeling that a support network exists and help is available from it.

**Instances** This pattern is utilized to arrange required knowledge support in an organization.

**Process Connection** Project management, competence development.

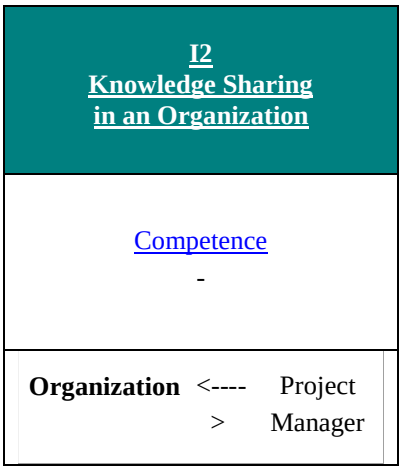
## Justification

**Basic Idea** To arrange continuous and project-specific support for projects.

**Instances** This is a structuring pattern and the instances are given for the actual patterns (KSP02 and KSP03).

## 15.4 KSP02 Named Experts

Dimensions and Knowledge Flow:




**Problem**

An organization not knowing how to make available the knowledge and experiences of some of its members to benefit many projects and the whole organization.

**Initial Context**

An organization having a need to make the existing experience of people more easily available to the whole organization.

**Roles**

*Organization* represented by some managerial actor responsible for competence management, and a *Project Manager* needing knowledge support from the organization.

**Forces**

Establishing an official support person / expert network gives visibility to the knowledge sources and offers better possibilities to utilize those. To share knowledge in an organization, the employees need to know what kinds of knowledge sources are available.

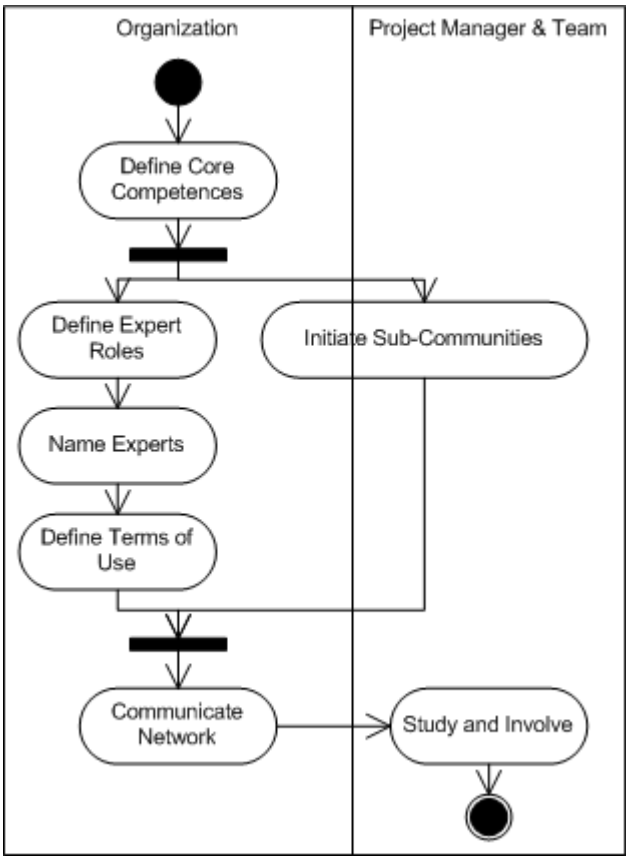
Pulling together persons having similar kinds of problems (like project managers) can give rise to new unofficial support groups in the organization. The fellow employees most probably have already solved some of the problems you have.

The role of an unofficial organization should not to be underestimated and it should be treated as a possibility in giving this kind of support.

Through cooperation with identified experts it is possible to share tacit knowledge in an organization.

Some general rules of using the support are required to avoid a situation where a person tries to delegate his or her tasks to others.

**Solution**




Implement the following actions:

1. *Define core competences* and other important competences in the organization. (See also Assured Resources, KSP19.)

2. *Define expert roles*. Define what competence areas are important enough for naming an expert. Define also what the role of an expert means in practice. The expert role here is a line management role, not project specific role.

3. *Name experts*. Assign the expert role to certain persons in the organization. In very small companies, think of the possibility of using externals for support roles as long as one's own experts have achieved an adequate competence level.
   Note that a person named to an expert role does not need to know everything, except the person needs to be skilful in acquiring new knowledge when required. Remember also the need to train experts.

4. *Define terms of use*. Define general rules how to utilize these experts.

5. *Initiate Sub-Communities*. Support the emergence of positive unofficial co-operation. (See Initiated Communication, KSP20.)

6. *Communicate Network* and *Study and Involve.* Assure that people know the existence of named experts and supported sub-communities. Make it possible to be involved in those.

**Resulting Context**

Project teams and especially Project Managers in the organization knowing that knowledge support exist and help can be got from named experts.

**Instances**

This pattern is utilized once to arrange this support in an organization and after that always when it needs to be redefined (e.g. changes in core competences).

| | |
|---|---|
| | One potential pitfall is that experts are named but they do not have any time to act in this role while being 100 % assigned to their own projects. |
| **Process Connection** | Human resources process, competence development. |
| **Related Patterns** | Domain Expertise in Roles<br>http://users.rcn.com/jcoplien/Patterns/Process/section7.html |

## Justification

| | |
|---|---|
| **Basic Idea** | To assure expert support network that can be used when needing support in decision making situations. |
| **Positive Instance** | Sharing expertise. Deeply embodied knowledge requires a strong interactional human-to-human processes through which the expertise is triggered and shared (Fitzpatrick 2003, pp. 106-107). Expertise-sharing networks Here expert networks help to locate individuals who possess needed expertise and who can facilitate sharing that expertise (Tiwana and Bush 2005). |
| **Negative Instance** | In project Alpha, availability of a well communicated expert network would have helped especially in the problem P3 Failure in Providing Organizational Support. |

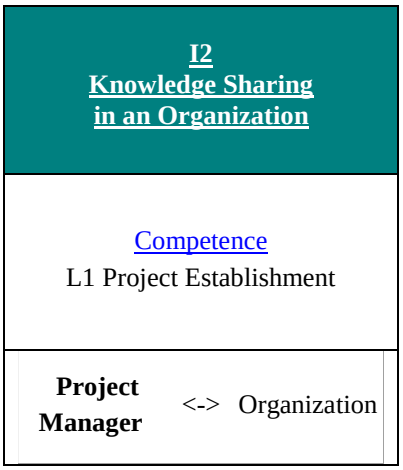
## References

Fitzpatrick, G. (2003). Emergent Expertise Sharing in a New Community. In Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise - Beyond Knowledge Management. MIT Press, Cambridge, Massachusetts, London, England.

Tiwana, A. and Bush, A. A. (2005). Continuance in Expertise-Sharing Networks: A Social Perspective. IEEE Transactions on Engineering Management. 52(1), 85-101.

## 15.5 KSP03 Assigned Experts

Dimensions and Knowledge Flow:




**Problem** A project team is missing certain competence areas that are required in a project.

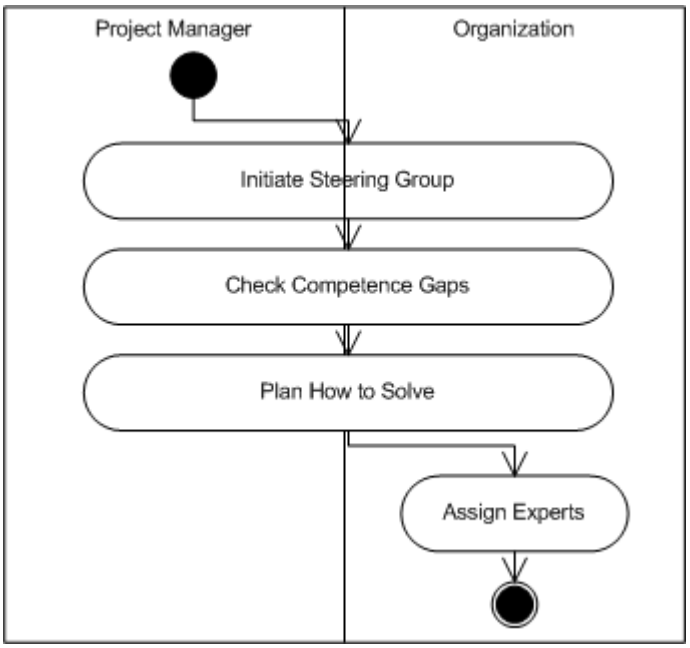
**Initial Context** Missing competence areas have been identified (e.g. Improved Competences, KSP18).

**Roles** *An Organization* represented by e.g. *Director of Projects* (head of the project office) together with the *Project Manger*.

**Forces** Good project managers are very valuable resources. The ones already having that status must be supported to avoid burn-out and the others must be trained to be good project managers.

Pressures for the project managers are quite high in many projects. Project managers have much power and need to make difficult decisions with limited amounts of knowledge. To have satisfied project managers with growing skills of software development projects, some kind of knowledge support must be arranged. This is also a good way of supporting project managers' learning-by-doing.

**Solution**




1. *Initiate steering group*. Define and assign a steering group for the project including persons having adequate authority and knowledge to support the project manager.
2. *Check Competence Gaps.* Look through the earlier identified competence gaps (e.g. Improved Competences, KSP18). Check also that the following perspectives have been covered:
    - Technology knowledge support
    - Process knowledge support, also quality assurance roles.
    - Project management knowledge support
3. *Plan How to Solve.* Decide, how the gaps and missing experts are to be found (in-house, external etc.). Utilize also the Named Experts (KSP02) when reasonable. Check availability of required persons.
4. *Assign Experts.* Name the experts for the project or give permission to utilize external experts. The assignment can be part time or full time. Communicate the assignments so that the selected experts and the project team knows about those. Note, that permission to purchase external expertise could have been taken also earlier e.g. for subcontracting.

Each project team member also has an unofficial, private contact network. That network is encouraged to be used noticing the information security etc. limitations (See My Network, KSP21).

**Resulting Context** Project teams and especially Project Managers in the organization have the feeling that a support network exists which can help.

**Instances** This pattern is utilized once to arrange this support for a new project and after that always when it needs to be redefined.

One potential pitfall is that experts are named but they do not have any time to act in this role while being 100 % assigned to other projects.

**Process Connection** Project management process

| | |
|---|---|
| **Related Patterns** | Mentor<br>Manns, M.L. and Rising, L. (2005). Fearless Change: Patterns for Introducing New Ideas. Pearson Education Inc., Addison-Wesley, 192-194.. |

## Justification

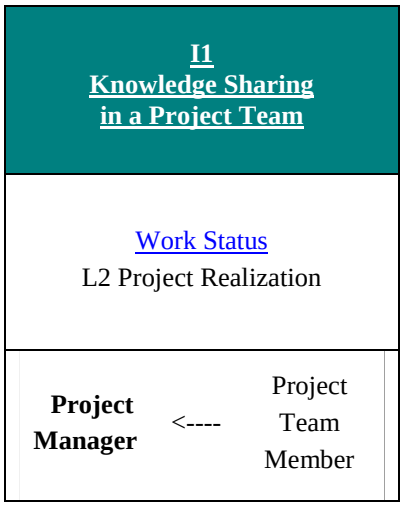
| | |
|---|---|
| **Basic Idea** | To establish a support network for a single project that may not have all the required competences in the project team. |
| **Positive Instance** | Sharing expertise. Deeply embodied knowledge, requires interactive human-to-human processes through which the expertise is triggered and shared (Fitzpatrick 2003, pp. 106-107). Thus also at the project level certain support/expert roles can contribute in competence that is not adequate in the project team already. |
| **Negative Instance** | In project Alpha, availability of defined expert persons to support the project would have helped (problem P3 Failure in Providing Organizational Support). |

## References

Fitzpatrick, G. (2003). Emergent Expertise Sharing in a New Community. In Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise - Beyond Knowledge Management. MIT Press, Cambridge, Massachusetts, London, England.

## 15.6 KSP04 Trust or Check

Dimensions and Knowledge Flow:




**Problem** A project manager is unsure about how to follow a project team member's progress in a project.

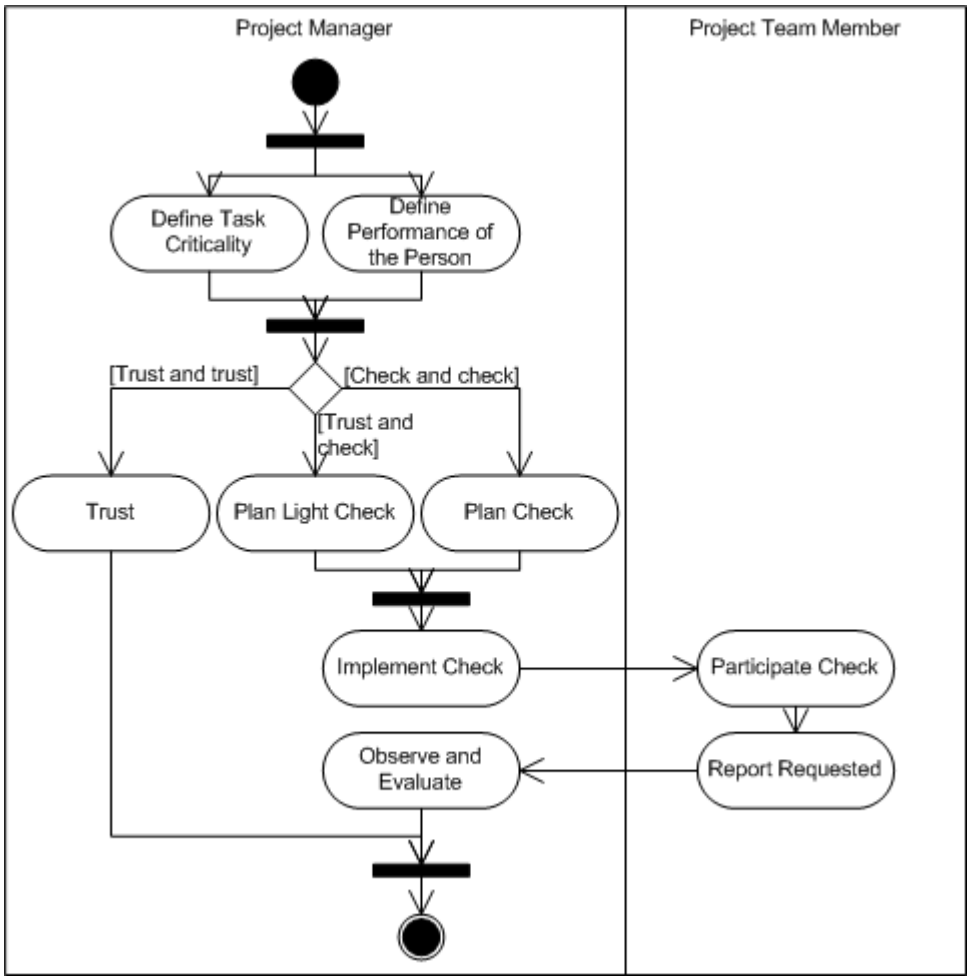
**Initial Context** The check activities here are introduced in addition to the normal quality assurance activities defined in software development processes. Here it is assumed, that normal document reviews, code reviews and testing are implemented.

**Roles** *A Project Manager* managing the project. The project manager is the key actor in this pattern. *A Project Team Member*, normally a Software Engineer, implementing tasks in the project.

**Forces** The results of the work of a project team member implemented in a project are visible normally only when some testable and/or visible artifact is produced. In some cases this may result in a situation, where a project team member implementing a single assignment behaves as if everything would be fine and the delays or problems are visible only after the deadline of the assignment. The purpose of this pattern is to help to have an understanding of possible problems as early as possible.

Constant checking and surveillance by the project manager takes time and may affect negatively the team spirit. It is not reasonable always to follow all project team members and their progress at a detailed level. Some kinds of criterion is required to determine when more follow-up might be required and when not.

**Solution**




1. Define trust or check requirements:

    i. *Define Task Criticality*
    Critical: check
    Not critical: trust

    ii. *Define Performance of the Person*
    Do you know well enough the performance of the Project Team Member implementing the task?
    Known, good performance: trust
    Known, poor performance: check
    Unknown: check

2. Decide the approach based on: i & ii. Plan required check activities. See e.g. section Practices.

    - Trust & trust: *Trust*, no additional check activities are required. Stop implementation of this pattern here.
    - Check & trust or trust & check: decide if you need any additional check activities. *Plan light check* activities.
    - Check & check: *Plan check* activities.

3. *Implement check* activities and follow the results (including also: *Participate check, Reports requested* and *Observe and evaluate*.
    Be open to the team member, tell him/her that because you are new to each other or because it is a very critical assignment, you need more information regarding the progress and thus will implement these checking activities together with this person

(or ask some other team member to do those with this person).

**Resulting Context**

After implementing these kinds of checking activities, the project manager knows the project team member better and can, in the future, either trust easier or knows what kind of checking activities are required. One result can also be that the person is found not to be suitable for these kinds of assignments.

The potential negative results of the surveillance must be followed. See also the potential pitfalls.

**Instances**

This pattern should be used always when assigning a new project team or when assigning a task (must be decided for all project team members participating in the task implementation). In many cases the pattern implementation is finished after deciding that a person can be trusted and the normal quality assurance activities in the project are enough.

Potential pitfalls:

- Checking activities starting to take too much time.
- Team spirit changing to negative because of detail level surveillance.

**Process Connection**

Supports Project Management process, project realization.

## Practices

Examples of recommended practices for check activities (see step 2 above):

- Establish good communication relationship to openly discuss about possible problems.
- If reasonable, divide the assignment into smaller subtasks with clearly defined results. Follow up so that the results are achieved.
- Use pair programming with a pair that you can trust and has the required competence.
- Have an unofficial preliminary review/testing during the task to have better understanding of the situation and the progress.
- Have someone else test the code/results produced by the team member.

## Justification

**Basic Idea**

To have basic criteria to estimate how progress of a single project team member should be followed during an assignment.

**Positive Instance**

The criteria included two main points: the criticality of the task and the track record of the project team member in earlier projects. These have been selected based on the interviews of project managers at the organization being studied. Also some other ones could have been selected, but these two together seemed to form a very effective and simple set.

The actual practice examples are just to give idea of possibilities. The user of this pattern is welcome to invent better practices.

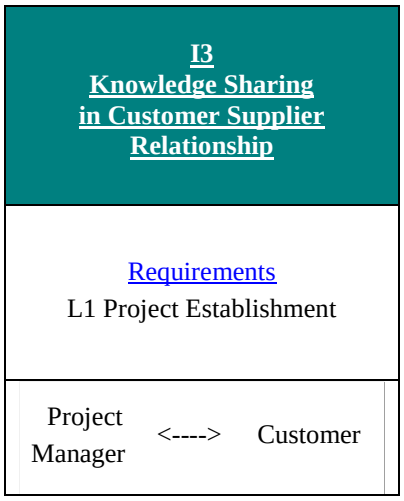
**Negative Instance**

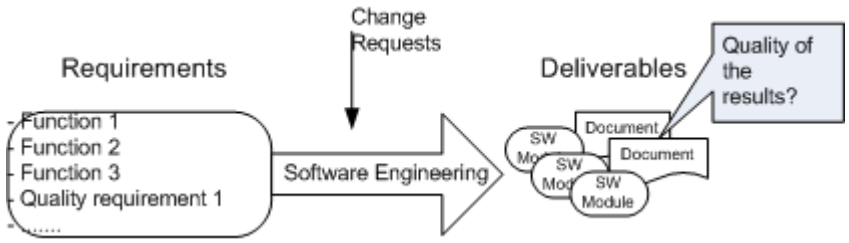
Had this pattern been in use in the project Alpha, the situation that occurred when poor results were found when a project team member left the project could have been discovered earlier. In project Beta, there was also a project team person that

could have been followed more intensively to assure the right results. In this latter case, however, the not so critical tasks were intentionally selected to the person so the lack of follow-up was not critical. This is one example of a possible practice used when there is a newcomer whose performance in general is not known by the project manager.

## 15.7 KSP05 Shared Understanding

Dimensions and Knowledge Flow:




**Problem** The aimed results of a project are not yet clear enough for all parties, especially between the customer and the supplier.

**Initial Context** A new software engineering project is planned or the aimed results of a current project are not clear enough to the customer and/or to the supplier.

**Roles** *A Project Manager* or alike, perhaps also a sales representative together with *a Customer Representative* define the expected results of the project.

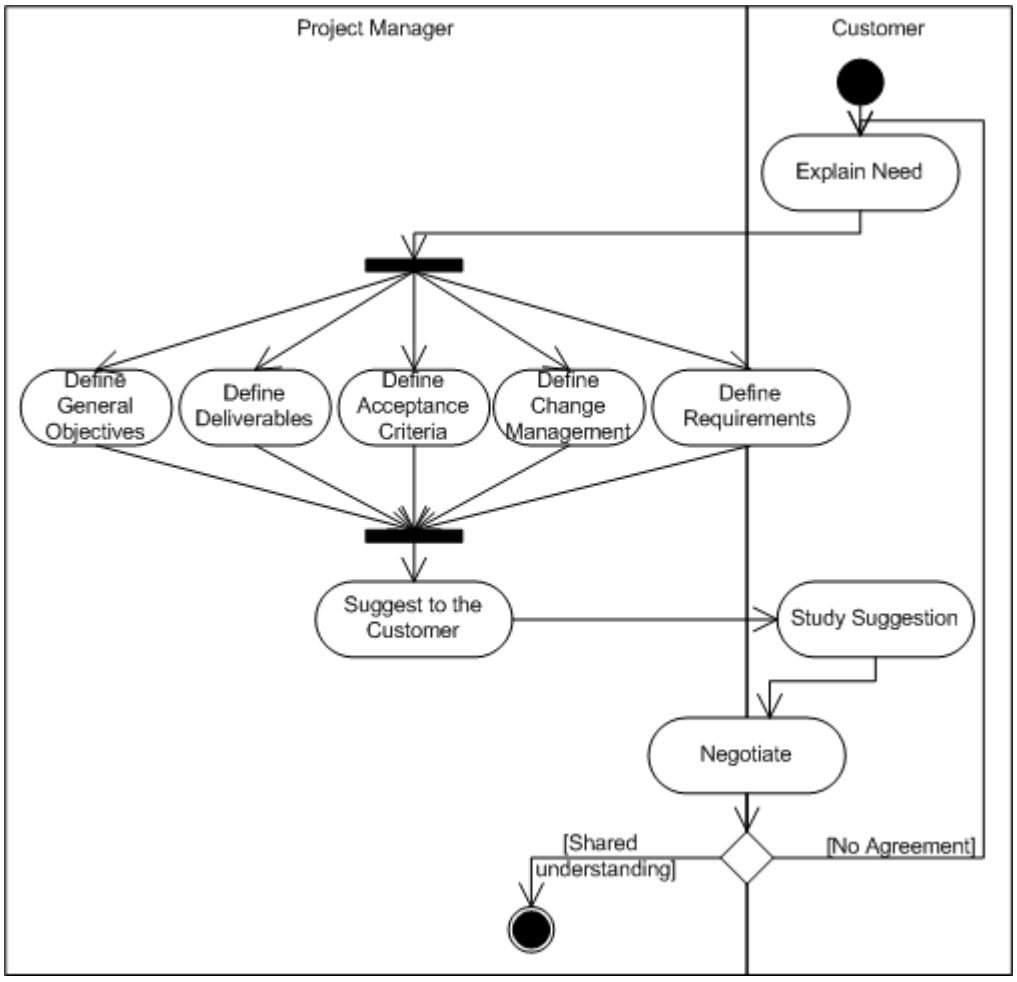
**Forces** In the establishment phase the project must first be defined in the form of required results. The results must be based on the customer's needs, and the supplier must have a clear understanding of what the aimed results are.

Detail planning of a project is not possible before the general objectives are defined. A more detailed requirements definition is possible only when you first have an understanding of the general objectives.

Normally, in practice, it is very difficult to define when a software engineering project can be closed. Not having well defined acceptance criteria might result in a customer and a supplier having totally different expectations.

Finally, change normally can not be avoided during a software engineering project.

**Solution**




To define and agree on the required results:

1. Customer *explains* the *need* (in practice, this continues parallel to the Step 2).
2. Supplier defines suggestion of :
    - *General objectives* for the project
    - *Deliverables*: what exact deliverables are required (software, documents, services,...)
    - *Acceptance criteria*: how the results are to be accepted and using what criteria? This could include a definition of the required quality level of the result or an allowed amount and type of defects, like "no blockers".
    - *Change management*: how changes to the defined requirements, results etc will be initiated, processed and approved.
    - *Requirements*: what functions etc. are required (See Discovered Bones, KSP06). Remember that requirements capture is in most cases iterative activity and requires involvement of several stakeholders. In the beginning of a project, however, some kind of early understanding is needed about the requirements or at least about the ways of capturing those.
3. Supplier makes a *Suggestion to the customer* and the customer *Studies the Suggestion*.

4. *Negotiate* to establish shared understanding of results for a project. If the shared understanding is achieved, then this pattern ends. If not, then the customer has to continue explaining the need. Shared understanding can mean an agreement of results, or an agreement that solution together (customer and supplier) is impossible.

Note that this is a simplified process defining the required results. Very often this is done parallel with the commercial discussions. Also the requirements definition can be done at this phase or later as a separate pre-study phase.

**Resulting Context**

Shared understanding created between the customer and the supplier about the aimed results of a project. Procedures and approval rights approved for processing changes. Clear acceptance criteria defined making possible objective judgment of the project readiness.

**Instances**

The pattern is aimed to be used always when project results are to be defined. Normally, this is in the beginning of a project or during the sales phase before a project.

One potential pitfall is that the acceptance criteria are not in sufficient detail to really be the basis when judging the project readiness.

**Process Connection**

Project establishment and sales.

**Related Patterns**

- Scenarios Define Problem
  http://users.rcn.com/jcoplien/Patterns/Process/section23.html
- Shared Vision
  http://www.publicsphereproject.org/patterns/print-pattern.php?begin=101
- Participatory Design
  http://www.publicsphereproject.org/patterns/print-pattern.php?begin=36

## Justification

**Basic Idea**

Instead of just starting to define requirements for a project, here, the aim has been to decompose the project target into different smaller parts. These are then negotiated with the customer to have shared understanding.

**Positive Instance**

This is a best practice that is found in an organization's project management process descriptions and in practice especially as a rule of three important aspects of defining a project: deliverables, acceptance criteria and change management. This practice together with one globally acting customer organization has also been confirmed. Coplien and Harrison (2005, p. 366) has defined patlet *Shared Clear Vision*, that has the similar purpose of creating first a shared vision about the system to be built. Also McCarthy and McCarthy (2006, pp. 11-20) highlight the importance of establishing a shared vision, or understanding as referenced here.

**Negative Instance**

In project Alpha one of the two main problems was problem P1: Weak Project Definition. This pattern would have helped e.g. through more explicit acceptance criteria (including the quality requirements) that was missing in project Alpha at the first part of the project. Also the creation of the shared understanding would have been more systematic when using this pattern.

## References

Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall.

McCarthy, J. and McCarthy, M. (2006). Dynamics of Software Development. Microsoft Press, Redmond, Washington.

## 15.8 KSP06 Discovered Bones

Dimensions and Knowledge Flow:


**Problem** Customer and the supplier not having common understanding of the requirements for the project.

**Initial Context** A new project is planned or about to be started or a separate pre-study including requirements definition is about to start. A preliminary understanding of project objectives exist (Shared Understanding, KSP05).

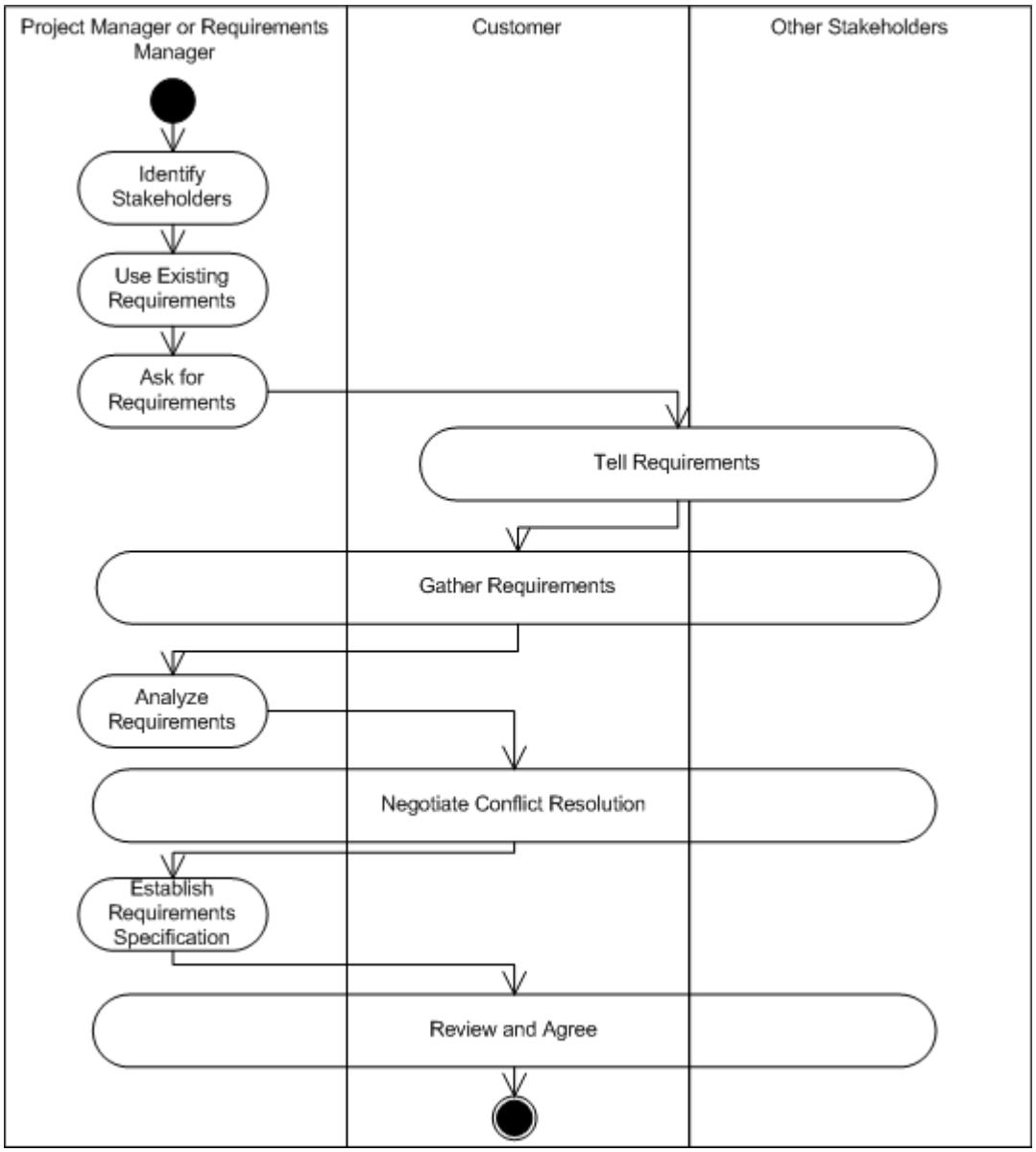
**Roles** *A Project Manager* or a Requirements Manager with the project team define requirements for the project together with the *Customer Representatives*.

**Forces** Requirements defined and shared with the customer are the representation of the customer and other stakeholders' need. The set of defined requirements guide the implementation of a project and act as a "skeleton" for the project linking different phases and products together. Or, like Clements and Northrop (2002, p. 110) say: "Requirements Engineering is not just an upfront activity but rather has ramifications over the entire development and maintenance effort."

In the establishment phase of a project the requirements for the project must be well established and shared between the customer and the supplier. Requirements are required also for other than pure software development projects. For example, integration projects have project requirements in addition to the requirements of the product to be integrated.

Depending on the planned commercial model the definition of requirements can be general (detail definition to come later) or as detailed as possible.

**Solution**




A general (normally iterative/incremental) procedure for defining requirements for a project is the following. See also section Practices for references to different approaches.

1. *Identify Stakeholders*. Define who benefits in a direct or indirect way from the system which is being developed or is affected some way from the system.
2. Check if you can use already existing requirements (See: Not Wasted, KSP23)
3. *Ask for Requirements, Give Requirements, Gather Requirements.* Collect requirements and document those with the source of the requirements.
4. *Analyze Requirements*. Check for possible conflicts or missing requirement details. Update the requirements based on findings.
5. *Negotiate Conflict Resolution*. Negotiate resolution to possible conflicts and collect possible missing details.
6. *Establish Requirements Specification.* Document requirements as a requirements specification (document, database etc.). Check that an adequate detail level has been achieved. Define which requirements can not be made specific enough and have those as items in risk follow-up.

7. *Review and Agree.* Review requirements with all relevant stakeholders and agree upon the baseline for requirements.

**Resulting Context** Well (or adequately) defined requirements to be the basis of the project and change management in the project.

**Instances** To be used always when defining requirements for a project.

One potential pitfall is that the commitment will not be forthcoming from the customer. In practice, this would mean not having adequate shared understanding between the customer and the supplier. This could result in a project that is difficult to close.

**Process Connection** Project Management - Project establishment.

## Practices

Requirements definition can be implemented using different approaches. The general practice has been defined above. More detailed examples could be examined from different software engineering approaches. In addition, the agreement type and status between the supplier and the customer may give some freedom or restrictions. In the following chapters, two different approaches have been introduced for requirements definition: traditional and agile approach.

Traditional Approach

OMT++ (Jaaksi et al. 1999) is used here as an example of the traditional, plan driven approach. There, a project is started with the phase *Requirements Capture*.

*Requirements Capture* phase collects raw requirements from various sources (several different stakeholders) and documents them explicitly (after preliminary analysis) as requirement statements (Jaaksi et al. 1999, p. 9). Use cases have been used as one very important tool for this phase. Like Jaaksi et al. (1999, p. 12) says: "On one hand, end users and software developers must be able to discuss the requirements and form a common understanding of what kind of system will be developed. On the other hand, software developers need a deeper and more detailed understanding of the functionality of the system." Use cases have solved this successfully for them.

Even though projects are normally implemented incrementally the main effort of requirements development and documenting is in the beginning of the project when the list of requirements is also frozen and after that changes are approved based on processed change requests.

Agile Approach

Scrum (Schwaber and Beedle, 2002) has been used here as an example of an agile approach to defining requirements. In Scrum the requirements are frozen only for a certain *Sprint*, a thirty-day iteration. The requirements are collected to a list called *Product Backlog*. This list is constantly prioritized and managed only by a *Product Owner*. For every Sprint the requirements to be implemented are prioritized by the Product Owner. Anything that represents work to be done in the project is added to the Product Backlog list. It is a list of all features, functions, technologies, enhancements, and defect fixes that are required in the future releases. It originates from many sources and the contents of it are discussed with many shareholders, including the customer. (Schwaber and Beedle, 2002, pp. 3-9 and 33).

In Scrum the Product Backlog is the list of requirements for a project (or product) of which the highest prioritized ones are implemented in the next Sprint and frozen for the duration of the Sprint. The solution given in this pattern should then be applied as an iterative procedure, selecting the highest priority topics from the Product Backlog list for each Sprint. This is not easy and not always possible to implement e.g. in fixed scope and price projects. Then,

some semi agile solution might be required to have the project adequately agreed on at the beginning giving the basis for scope and cost management.

## Justification

**Basic Idea** To collect input from all stakeholders to know what requirements the project has.

**Positive Instance** Based on existing, well known best practices. Best practices e.g. from Sommerville and Sawyer (1997) and CMMI (Chrissis et al. 2003) have been the basis when defining this pattern. The best practice definitions have been a bit too detailed and too fragmented so here the highlights of those have been applied to give the general understanding of the required procedure.

**Negative Instance** Luckily no example was found in the target organization of a case of not having somehow defined requirements. In some cases there have been problems in having a commitment from the customer to the detail requirements. One of the reasons for that could be that customer representatives think that their hands will then be tied too tightly, but at the same time the supplier project manager has difficulties in managing the project scope.

## References

Chrissis, M.B., Konrad, M. and Shrum, S. (2003). CMMI: Guidelines for Process Integration and Product Improvement. SEI Series in Software Engineering, Addison-Wesley.

Clements, P. and Northrop, L. (2002). Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley.

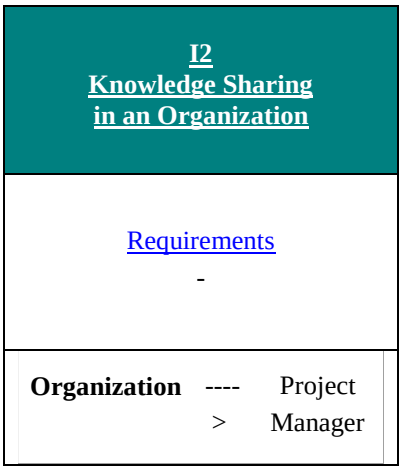
Jaaksi, A., Aalto, J-M., Aalto, A. And Vättö, K. (1999). Tried & True Object Development: Industry-Proven Approaches with UML. SIGS Books, Cambridge University Press.

Schwaber, K. and Beedle, M. (2002). Agile Software Development with Scrum. Prentice Hall Series on Agile Software Development, Upper Saddle River, New Jersey.

Sommerville, I and Sawyer, P. (1997). Requirements Engineering: A Good Practice Guide. John Wiley & Sons, Chichester, UK.

## 15.9 KSP07 Reference Requirements

Dimensions and Knowledge Flow:




**Problem**

No knowledge about what requirements definitions might already exist and how to find those as potential reusables for projects.

**Initial Context**

An organization wanting to establish systematic reuse of earlier defined requirements.

**Roles**

*Organization* represented by a person in a role responsible for developing software engineering, products etc. The right role varies much between organizations.

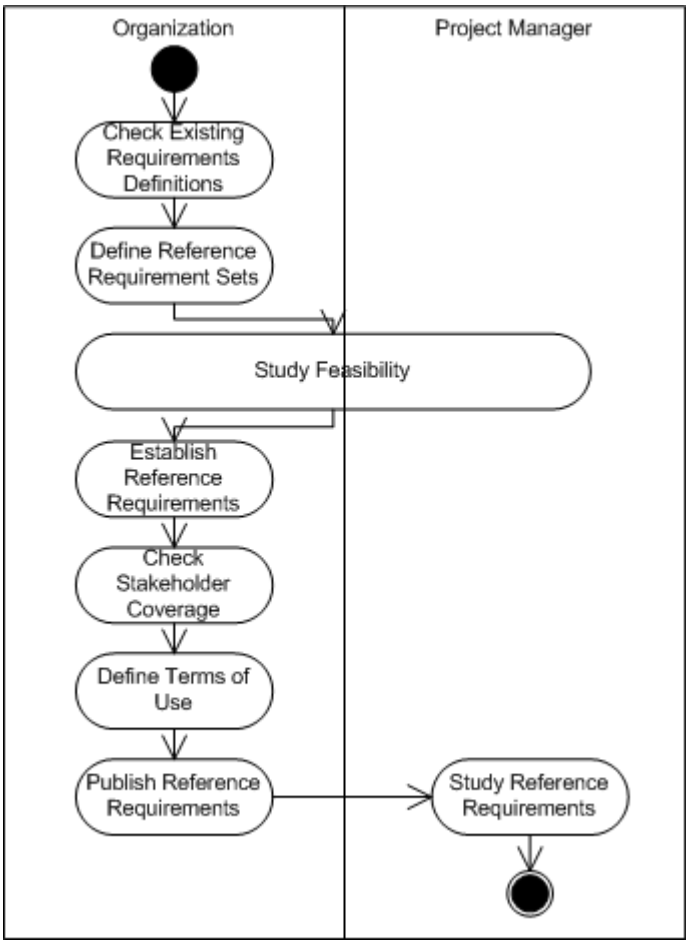
*A Project Manager* or Requirements Manager defining requirements.

**Forces**

Requirements for a project are normally defined under strict time pressure. If then, some earlier defined and tested sets of requirements could be used, it would shorten the required time and also result in more quality requirements compared to the situation where all requirements have to be defined starting from none existing.

Normally, there are several earlier projects etc. where requirements have been defined. It is not easy to find the right one when being in a hurry. Also, in most cases, the requirements have been improved in other projects but having many project-specific features.

**Solution**




To establish possibilities for reusing existing requirements do the following:
(See also product line based approach from the Other Practices section.)

1. *Check Existing Requirement definitions.* Define what kind of requirements exist and what could or should be available for reuse based on business needs.

2. *Define Reference Requirement Sets.* Define sets of reference requirements that might be collected. Define owners for those having the responsibility to guide the use of the requirements.

3. *Study Feasibility*. Together with sales and Project Managers, study the feasibility of the planned Reference Requirements sets. Main criteria is adequate future need.

4. *Establish Reference Requirements.* Start establishing reference requirements sets. Use existing requirements or initiate a separate project to establish new reusable reference requirements. (Improve the set of requirements over time, see Not Wasted, KSP23).

5. *Check Stakeholder Coverage.* Check that all required stakeholders are notified including authorities (e.g. product safety and liability requirements).

6. *Define Terms of Use.* Define terms of use for each set of requirements. Notice also possible IPR restrictions and warnings.

7. *Publish Reference Requirements.* Publish the sets of reference requirements, terms of use and contact persons. Pay attention to easy finding and use.

8. *Study Reference Requirements*. Project Managers need to become familiar with the requirements at such a level that they know what those are and when they

could be applicable.

| | |
|---|---|
| **Resulting Context** | Clearly defined sets of reusable reference requirements existing ready to be used by projects and maintained by the organization. |
| **Instances** | To be utilized once to initiate reference requirements sets and after that for each new requirements set.<br>E.g. in software production for mobile phones there could be sets like requirements for compliance to certain operator requirements, certain market area requirements etc. Requirements could also be related to certain products of the company.<br>One possible pitfall is to introduce a set of requirements that will not be utilized. -> Step 3, Feasibility Study is very important and should involve people having good business opportunities and market understanding. |
| **Process Connection** | Software engineering, requirements definition. |

## Justification

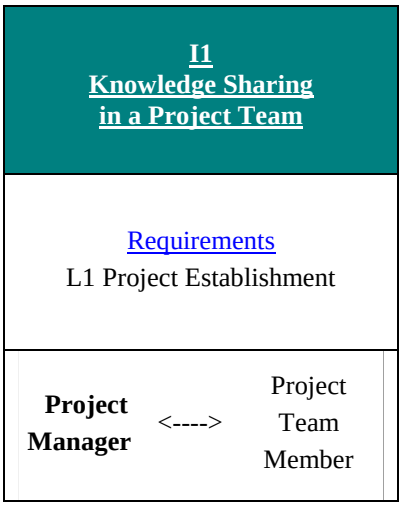
| | |
|---|---|
| **Basic Idea** | To introduce sets of earlier defined requirements to be available for reuse in new projects. |
| **Positive Instance** | In product-line based software development, the common requirements together with identified variation points are one important part of the reusable core assets (Clements and Northrop 2002). Product-line approach aims at building many similar kind of products based on the product line core assets. |
| **Negative Instance** | Not valid here. A project can be made also without using any earlier defined requirements |

## References

Clements, P. and Northrop, L. (2002). Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley.

## 15.10 KSP08 Created Skeleton

Dimensions and Knowledge Flow:




**Problem** Project team members having only very general knowledge about what the results of a project should be.

**Initial Context** A new project is about to start and a project team is assigned to it. A preliminary understanding of project objectives exists (Shared Understanding, KSP05).

**Roles** *Project team members* and *a Project Manager* or a person in the role of Requirements Manager.

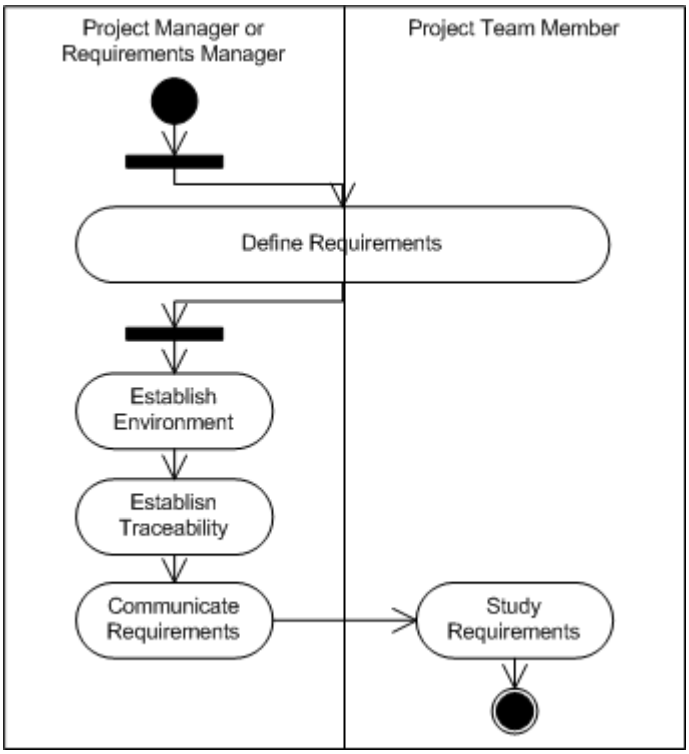
**Forces** Requirements are a structured way to introduce what is required from a project. One of the origins for requirements is the customer and what the customer wants from the project. Requirements thus represent a customer need/problem that is to be solved with the results from the project.

Requirements are linked to all parts of the project implementation trough the traceability information. Those are a sort of a skeleton for a project.

In addition to agreeing upon the requirements with the customer, those requirements must also be communicated effectively among the project team to really utilize this "skeleton" and to assure correct final results. Requirements specification represents the first real understanding of what results the project should expect.

A good way to communicate the requirements in a project team is to involve as many persons from the project team as possible to define the requirements.

**Solution**




1. *Define Requirements.* As much as possible, use all team members for defining requirements. Define the first baseline of requirements. (See: Discovered Bones, KSP06)

2. *Establish Environment.* Establish the working environment for requirements management including storing, and initiate the use. In a small project this could be an Excel sheet of requirements and in a bigger project a dedicated database.

3. *Establish Traceability*. Use the main requirements as a basic unit when defining what to implement when and where and what to test when. Describe the traceability chains in reasonable way. If possible, utilize tools to automate the linking.

4. *Communicate Requirements* & *Study Requirements*. Assure that project team members have the correct and adequate understanding of the requirements.

**Resulting Context**

Requirements for the project are well known among the project team making cooperation possible and allowing separate sub teams to work with different tasks. A basis has been created to trace information.

One potential pitfall is that project team members are not having adequate possibilities to be involved in defining/accepting requirements resulting in a situation where they do not understand the reason for all requirements. This might affect their commitment to the project and to the implementation.

**Instances**

To be used always when a new software engineering project is established.

One potential pitfall is a case where the requirements are defined by the project team and no proper commitment is asked for or received from the customer. This could result in project work results that the customer can not approve. -> importance of the Discovered Bones, KSP06 with strong customer participation.

**Process Connection**

Requirements Management

**Rekated Patterns** Surrogate Customer
Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 116-117.

## Justification

**Basic Idea** To have well managed requirements and traceability so that everyone in the project team knows what they are and can cooperate based on them.

**Positive Instance**

This pattern is built based on general requirements (stated in literature) for having standard processes, requirements document, change management, traceability etc. added with the requirements to involve different stakeholders. Here, the interfaces inside the project team have been made clearer in the form of this knowledge sharing pattern. The importance of that interface is too easily missed when emphasizing the importance of understanding the customer and its needs.

The literature normally refers to the stakeholders in general, not referring very much to the project team's internal communication. Sommerville and Sawyer (1997, p. 37) define the requirements document to be a vehicle for communicating the requirements to different stakeholders. CMMI (Chrissis et al. 2003) introduces also best practices for requirements development and management. It defines several specific practices, not directly solving the issue of knowledge sharing or communication in a project or in the project team, but through generic practices also this element is visible. Especially the generic practice Identify and Involve relevant stakeholders make the need for knowledge sharing visible.

This practice is taken from the processes of the studied organization.

**Negative Instance**

In project Alpha, requirements were defined at the general level, but those did not guide the implementation of the project and requirements management did not exist in any formal way. While not having well established requirements management, one of the many resulting problems was that the distant project team members (problem P13 Communication problems between the main project team and the distant members) did not know how their work was related to the targets of the project.

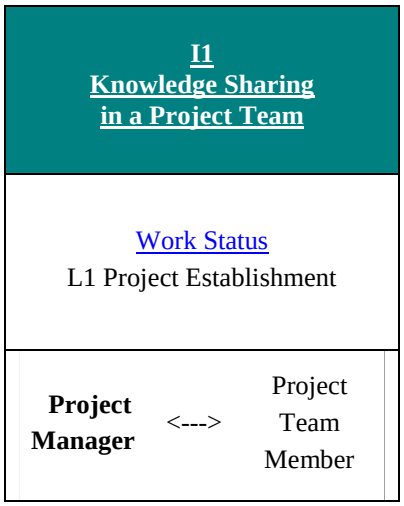
## References

Chrissis, M.B., Konrad, M. and Shrum, S. (2003). CMMI: Guidelines for Process Integration and Product Improvement. SEI Series in Software Engineering, Addison-Wesley.

Sommerville, I and Sawyer, P. (1997). Requirements Engineering: A Good Practice Guide. John Wiley & Sons, Chichester, UK.

## 15.11 KSP09 Schedule Baseline

Dimensions and Knowledge Flow:




**Problem** A project manager can not start proper project progress follow-up, because she/he does not have defined tasks nor a schedule for comparison with the current status.

**Initial Context** A new project starting or just started.

**Roles** *A Project Manager* managing the project and *Project Team Members* implementing the project.

**Forces** Project Manager should constantly know what the real status of a project is compared to project objectives and plans. Without this knowledge managing a project would be like driving in a dense fog.

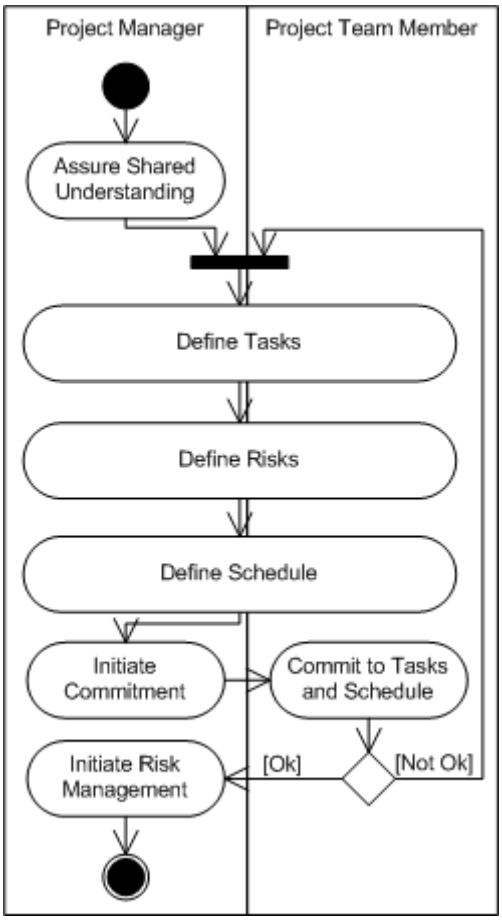
Risks should affect the planning and the project management later.

If bigger changes take place, it might be wise to revise the plan.

If it is clear that many changes will occur during a project, the plan should be looked through periodically and planning the early steps in more detail than the later steps.

The team needs to be motivated and committed to implement the project in the best possible way.

**Solution**




To have adequate understanding about what is required in the project and what is the "baseline" to be used when evaluating the current work status, do the following:

1. *Assure* that *Shared Understanding* (KSP05) regarding the results exist with the customer.
2. *Define tasks (*preferably together with the team) based on the requirements (see also: Created Skeleton, KSP08).
3. *Define risks* related to the tasks and the whole project.
4. *Define schedule* for the tasks (preferably together with the team). Plan the highest risk tasks to be implemented as early as possible.
5. *Initiate commitment & Commit to tasks and schedule.* Have commitment from the team. Be prepared to change the plan if a commitment is not received.
6. *Initiate risk management* procedures in the project.

In addition, some kind of formal approval is required for the project plan.

**Resulting Context**

A Project Manager running a project having clearly defined tasks and schedule to base the progress follow-up on. First risk identification is implemented and risk management actions are initiated. The project team has committed to the implementation of the project as planned. The project manager understands that when changes occur, it is possible to revise the project if required.

**Instances**

The Schedule Baseline should be created in the beginning of a project and always when

bigger changes requiring revision take place.

One potential pitfall is forgetting that a project plan has been made. The project starts following the plan, but when changes happen the plan is not updated anymore and can not support the project implementation and the progress follow-up.

**Process Connection** Project Management process, establishment.

**Related Patterns**

- Work Split
  Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 58-59.
- Named Stable Bases (integration perspective)
  http://users.rcn.com/jcoplien/Patterns/Process/section32.html
- Design Stance
  http://www.publicsphereproject.org/patterns/print-pattern.php?begin=44

## Justification

**Basic Idea** To create a plan and use that as the basis for progress follow-up.

**Positive Instance** Defining a project plan including scheduling is a basic action in a project. This pattern can be found in most of the projects in the studied organization and it is recorded into the processes of the organization.

It is also a specific goal to Develop a Project Plan in CMMI (Chrissis et al., 2003, pp. 413-422) as part of the project planning process area.

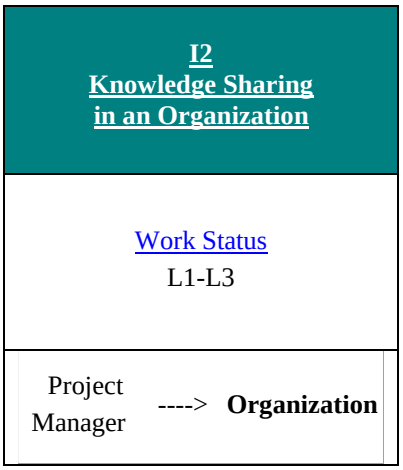
**Negative Instance** The project Alpha had a project plan when starting, but many changes took place during the project. The project was not revised during the first part of the project. When noticing the differences in understanding between the customer and the organization, a revision immediately could have helped.

## References

Chrissis, M.B., Konrad, M. and Shrum, S. (2003). CMMI: Guidelines for Process Integration and Product Improvement. SEI series in software engineering, Addison-Wesley.

## 15.12 KSP10 Known Status of Projects

Dimensions and Knowledge Flow:



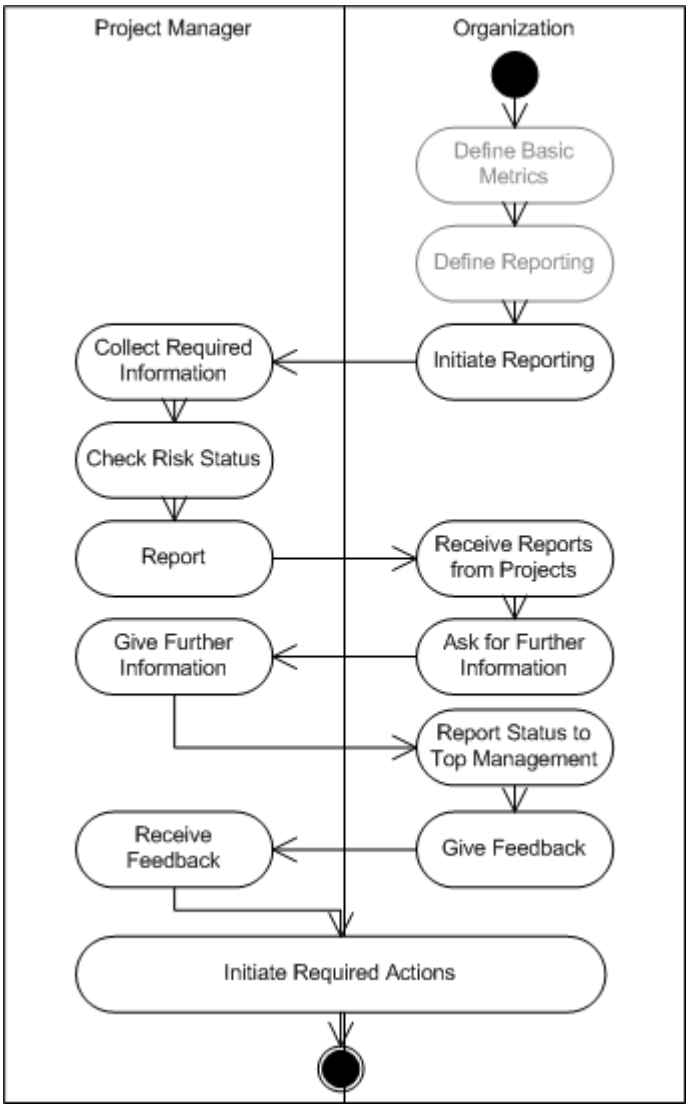

**Problem** Not knowing adequately enough the general situation of projects in an organization and thus no possibilities to manage the situation with several projects.

**Initial Context** An organization in project business having several ongoing projects. Projects having Schedule Baseline (KSP09) and Followed Progress (KSP25).

**Roles** *Organization* represented by a role Director of Projects (or alike) acting on behalf of the organization and collecting status information of projects and initiating organization level activities based on this information. *A Project Manager* reporting the status of a single project to the Director of Projects.

**Forces** To manage a project company, it is very important to know the status of projects in general and the main risks in detail. Existing projects compete for the same scarce resources in the organization and the resource pool should constantly be developed. In addition to managing single projects, it is also important to manage the multi-project situation. Adequate visibility is then required for the management into single projects to take required actions on the organizational level.

**Solution**




1. *Define basic metrics* to be used. This does not need to be done at every cycle.
2. *Define reporting* (how to be implemented, how often, etc.) This does not need to be done at every cycle.
3. *Initiate reporting*. Communicate the reporting need and guidance. Ask for reporting.
4. *Collect required information*. A Project Manager collects required metrics etc. information regarding to the project.
5. *Check risk status*. A Project Manager checks, and updates, if required, project's risk status and especially topics that might need escalation or support from the other organization.
6. *Report*. A Project Manager reports to the organization as defined.
7. *Receive reports from projects*. Director of Projects collects the reports from all projects.
8. *Ask for further information* and *Give further information*, if further information is required.
9. *Report status to top management*. Director of Projects communicates the status in projects to the other management.

10. *Give feedback* and *receive feedback* if any feedback has resulted from the discussions.

11. *Initiate required actions*. Initiate possible required corrective and preventive actions (by the project or by the organization).

**Resulting Context** A project organization having good knowledge of existing projects and possibilities to plan and manage the multi-project situation. The projects' risk level is known and possible actions can be initiated at the organizational level.

**Instances** The reporting first to be implemented (more effort taken for defining the metrics and reporting) and then automatizing as much as possible for the organization's standard reporting cycle.

Potential pitfalls:

- Project Managers not wanting to report anything -> Motivation, asking less but more important information.
- A Project Manager becomes passive in own proactive reporting and misses sharing information early enough about big risks or problems. -> Highlighting the importance of direct communication in addition to the formal reporting.
- This reporting might become so automatic, that not even the results are studied carefully and no actions initiated when required. -> Evaluating the effectiveness of reporting and initiating actions based on that.
- The reporting could work well even though the expected metrics would not serve the purpose. -> The usability and the effectiveness of the metrics and report questions shall be followed and improved.

**Process Connection** Multi Project Management at the organizational level.

## Justification

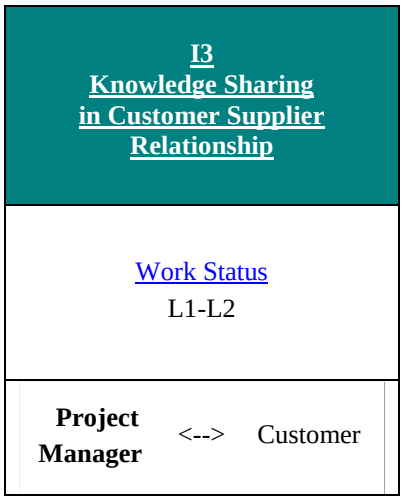
**Basic Idea** To understand the total situation in a project company by collecting standardized information from all projects.

**Positive Instance** A general pattern based on existing project reporting processes at the studied organization. Procedure and especially the content of reporting improved after the problems at the beginning of project Alpha.

**Negative Instance** This pattern could have helped in project Alpha. The organization could have found out earlier that the project has many big problems. In the organization there existed some kind of general reporting, but it was not systematic enough in this situation.

## 15.13 KSP11 Informed Customer

Dimensions and Knowledge Flow:




**Problem** Customer and supplier not having adequate common understanding about the project status and/or not communicating adequately about changes in both organizations affecting the project through interdependencies.

**Initial Context** Shared Understanding (KSP05) exists of the aimed results. A Schedule Baseline (KSP09) has been created and Progress is Followed (KSP25).

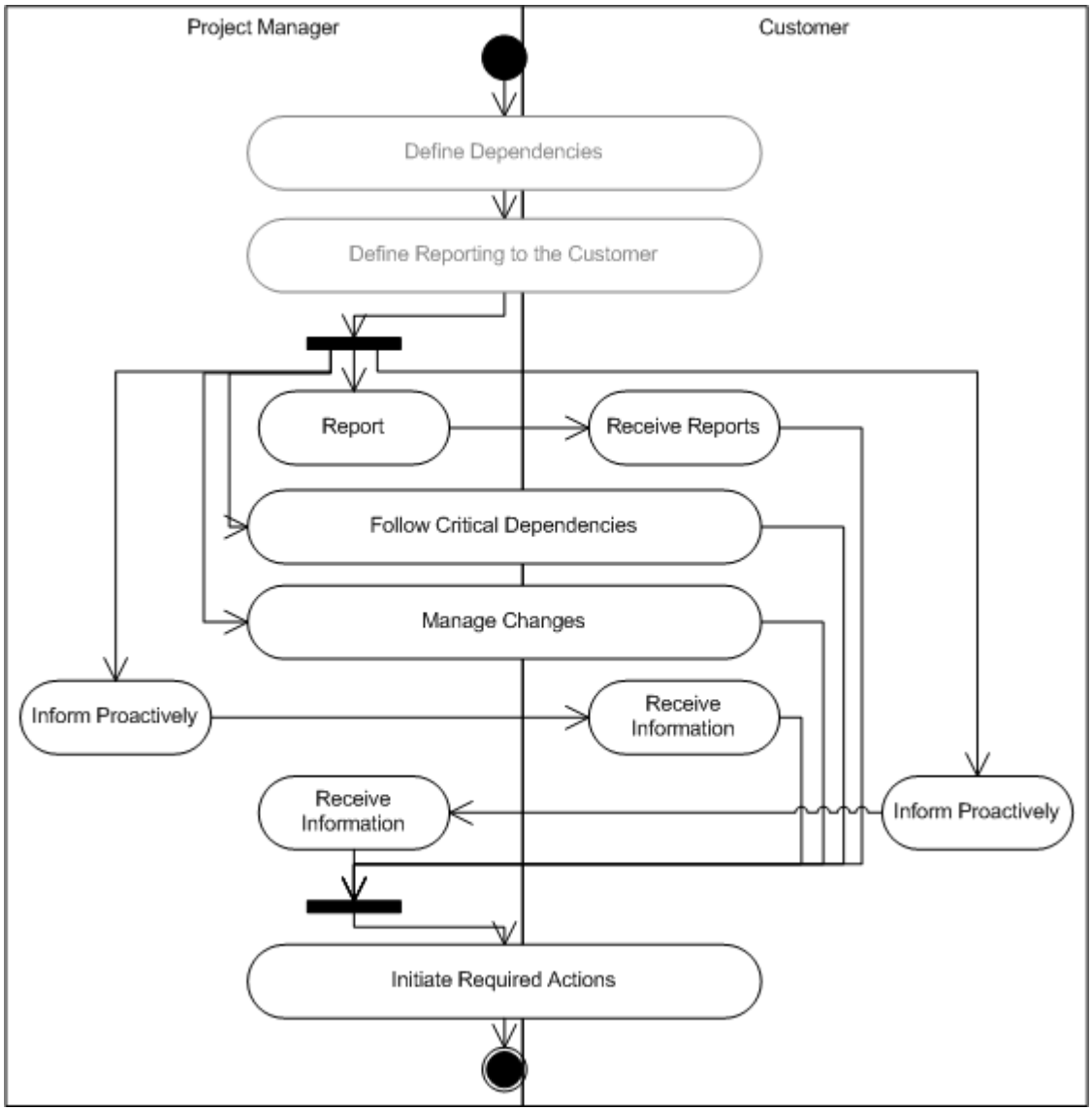
**Roles** *A Project Manager* managing a project and *a Customer Representative*.

**Forces** To cooperate successfully with the customer to create solutions to customer's needs a mutual understanding must exist about the work status of a project. The results and also the intermediate results of the project have been scheduled and may affect other activities in the organizations. Also the project may need something (e.g. certain software modules) from the customer and if the schedule of those changes, the project needs to revise the schedule.

It is not necessary that the customer knows everything that happens in the project but the customer should know the most critical matters adequately to trust the statements of the project manager and to follow the progress.

Open honest communication between the customer and the supplier creates a good basis for mutual trust. Of course, real actions and results need to support the messages and prove those correct.

**Solution**




1. *Define* critical *dependencies* in the project together with the customer. Notice especially intermediate results that the project needs or that some organization needs from the project at a certain deadline.

2. *Define* how the Project Manager *reports to the customer* and what metrics shall be followed together. For example, reporting according to the standard procedures of the organization with the exceptions requested by the customer. Notice that reporting may include written reports and e.g. steering group meetings with the customer.

3. *Report.* Implement reporting as planned.

4. *Follow critical dependencies*.

5. *Manage changes* together with a named customer representative.

6. *Inform Proactively* and *Receive Information*. For both, the customer and the supplier: inform as early as possible the customer / the supplier about high risks and problems.

7. *Initiate required actions* based on the information received.

This is actually a continuous activity which starts again from step 3. If there are bigger changes in the project, steps 1 and 2 might be also needed.

**Resulting Context** A project where both customer and supplier have adequate understanding of the project and its status including possible effects on other dependent activities.

**Instances** To be implemented for each new project. The reporting and follow-up continues throughout the project.

One possible pitfall is that an adequate level of open communication is not reached and one or both organizations try to hide things from each other. This is very possible especially if e.g. the customer does not want to invest any time in this or if there are big problems in the project and not courage enough to open the problem to the other party.

**Process Connection** Project Management, project establishment and realization.

**Related Patterns** Engage Customers
http://users.rcn.com/jcoplien/Patterns/Process/section21.html

## Justification

**Basic Idea** Supplier defines together with the customer, how the customer wants to follow the progress of the project. Then implementing the agreed follow-up and maintaining open communication in order to openly share also the problems.

**Positive Instance** This is a pattern that is based on existing project management processes and activities especially in successful projects (for example project Beta) in the studied organization. Expectation management (e.g. Boehm and Turner, 2004, pp. 155-156) is one target that this practice has. By including the customer in the project, and making the progress visible, the customer expectations can be balanced with open and realistic information regarding the project and its progress. Boehm and Turner (2004, p. 155) state that "the differences between successful and troubled software projects is most often the difference between good and bad expectations management."

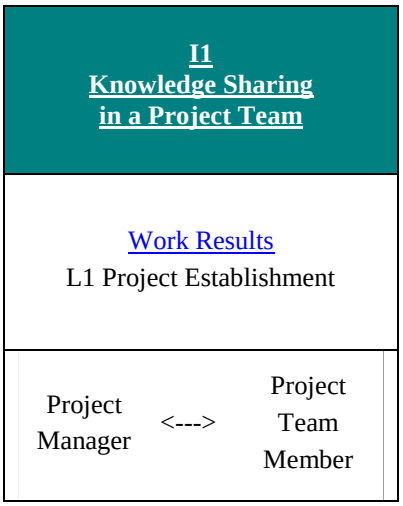
**Negative Instance** Project Alpha had a good informal start for customer communication but later when the different understanding of the customer and the supplier about the results was clear, the customer communication did not work so well. This pattern would not have solved the basic problem, but would have forced systematic cooperation and common follow-up of the project. Project Alpha did not have defined reporting etc. to the customer.

## References

Boehm, B. and Turner, R. (2004). Balancing Agility and Discipline: A Guide for the Perplexed. Pearson Education Inc., Addison-Wesley.

## 15.14 KSP12 Managed Versions

Dimensions and Knowledge Flow:




**Problem** Difficulties in sharing the (intermediate) work results in the project team.

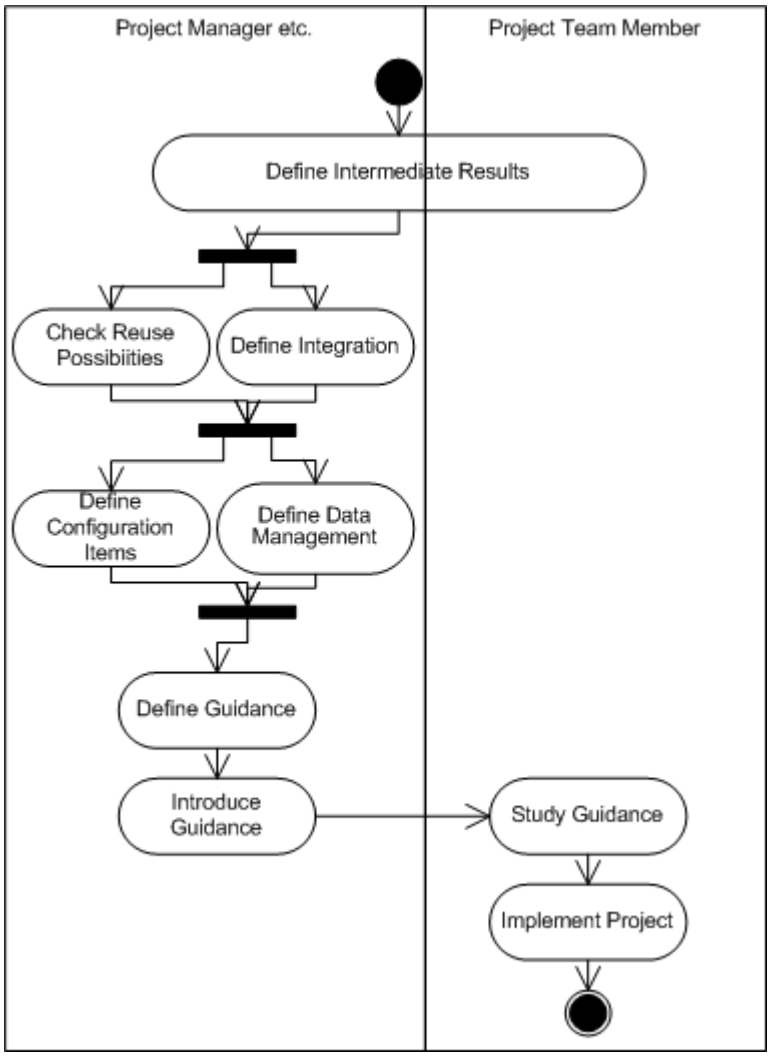
**Initial Context** A new project has been started and the deliverables required from the project are known (see Shared Understanding, KSP05).

**Roles** *Project Team Members* of a project. *Project Manager*, Release Manager or some other role. For simplicity here the role Project Manager is used.

**Forces** Project team members can not work fully separated. They must cooperate to achieve the results required in the project. Results from a project might be software code, documents, designs, etc. To work effectively the team must have ways to share this knowledge with each other and to know what the others need and what they produce.

Because parts of software need to be implemented by different persons, integration and the working of software modules together is critical.

**Solution**




To have the basic infrastructure existing for sharing work result knowledge in the project team the project manager (or another named person in the team) should do the following.

1. *Define Intermediate Results.* Based on project deliverables (see Shared Understanding, KSP05), define the required intermediate results.

2. *Check Reuse Possibilities*. Check if the work results from earlier projects could be utilized in this project (see Quickly Made, KSP13).

3. *Define Integration*. Define the principles of integration and required tools in this project. Initiate establishing required environment.

4. *Define configuration items* of this project and their requirements for configuration management.

5. *Define Data Management.* Define how (intermediate) work results are stored and shared in the project. Define and initiate also the use of a selected configuration management system. Define how to handle software code, documents, designs etc. And how to combine those to builds and releases. Initiate establishing required environment.

6. *Define Guidance*. Define required guidance for intermediate results, integration, configuration and data management.

7. *Introduce Guidance* to other project team members and possible other relevant stakeholders.

8. *Study Guidance* and *Implement Project* according to it.

**Resulting Context** Required intermediate results and how to find those are known in project team. Agreed integration and data management known and in use making possible cooperation and separate sub teams working with separate tasks.

**Instances** To be used when a new project is initiated. The utilization of the guidance continues till project is closed.

**Process Connection** Software engineering.

**Related Patterns**

- Named Stable Bases (integration perspective)
  http://users.rcn.com/jcoplien/Patterns/Process/section32.html
- Private World
  Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 46-48.

## Justification

**Basic Idea** To establish proper configuration management and integration.

**Positive Instance** The processes used and observed in many projects in the studied organization.

Specific goal: Establish Baselines at the Configuration Management process area in CMMI (Chrissis et al., 2003, pp. 160-164).

Specific goal: Prepare for Product Integration at the Product Integration process area in CMMI (Chrissis et al., 2003, pp. 372-377).

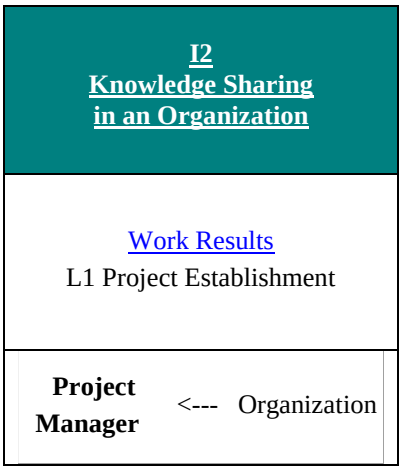
**Negative Instance** This pattern could have helped in the project Alpha. There a configuration management tool was used but no real guidance was given as to how to use it. For example, some newcomers added non-compilable code there making the work of others very difficult. Also missing clear definitions about how to use the database made the work of the persons on the remote site very difficult because they could not continuously ask the others about the status of materials in the configuration management system.

## References

Chrissis, M.B., Konrad, M. and Shrum, S. (2003). CMMI: Guidelines for Process Integration and Product Improvement. SEI series in software engineering, Addison-Wesley.

## 15.15 KSP13 Quickly Made

Dimensions and Knowledge Flow:



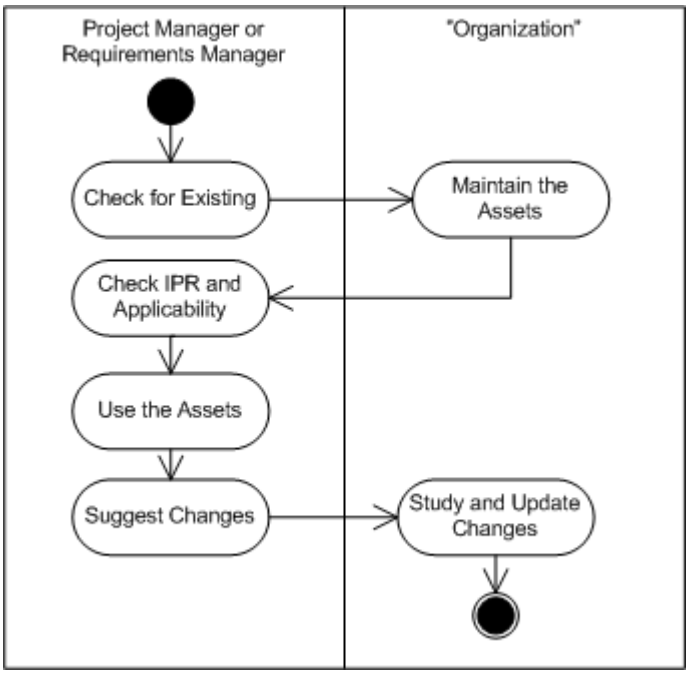

**Problem** Time available is very limited to have the work results implemented and having the defined level of quality.

**Initial Context** Some earlier similar kind of work results exist. A Reuse Approach (KSP26) has been defined and initiated.

**Roles** *A Project Manager* defining (intermediate) work results. *Organization* represented by a person being the named owner for the reusable assets.

**Forces** In most of the customer projects, the time-to-market is very critical. If, in such a time pressure, every work result is made "from scratch" there will normally not be adequate time for testing the work results. These together make it very reasonable to reuse earlier work results when possible. This can be done by reusing earlier ones or buying some produced by others. IPR related matters, however, must be very well considered.

**Solution**




To use existing work results:

1. When defining required work results *check for existing*: in the organization and external ones.
2. *Maintain the Assets*. Support projects how to utilize the existing assets. Study needs and decide maintenance and other activities.
3. *Check IPR and Applicability*. Check applicability of the potential assets. Check the IPR situation (availability for the project, possible needs for protecting own IPRs).
4. Check the terms of use and *use the* selected *assets* according to the terms.
5. *Suggest (and study) changes*. If changes are implemented to selected assets, check if those should also be implemented to the reusable assets.
6. *Study and Update Changes*. Study suggested changes and decide possible actions.

**Resulting Context** Shortened time-to-market and through earlier tested work results even better quality compared to starting from scratch.

**Instances**

To be used always when defining work results. Check if it is possible to reuse something.

One potential pitfall is reuse because of reuse and not because of business or effectivity reasons. This could lead to a situation where efforts are taken to establish reusable assets that will not have adequate return of investment.

**Process Connection** Software engineering

## Justification

**Basic Idea** To speed up the project implementation through using already existing and tested work

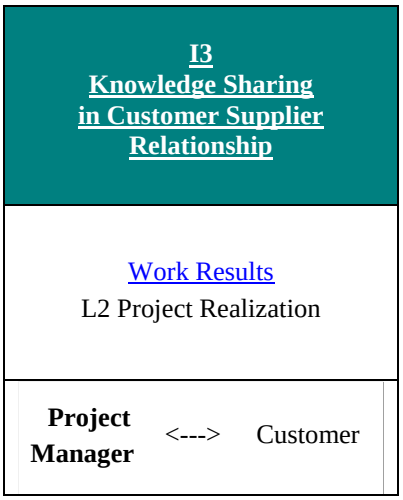
| | |
|---|---|
| | results. |
| **Positive Instance** | For example, the process for systematic software reuse by Jacobson et al. (1997, pp. 15-17). |
| **Negative Instance** | Not valid here. A project can be made also without using any earlier work results, but the amount of time to be used might be longer. If there will be similar projects / project results after the project, then it could be reasonable to think if some part of or all work results could be reused later. |

## References

Jacobson, I., Griss, M. and Jonsson, P. (1997). Software Reuse: Architecture, Process and Organization for Business Success. ACM Press, Addison-Wesley.

## 15.16 KSP14 Release

Dimensions and Knowledge Flow:



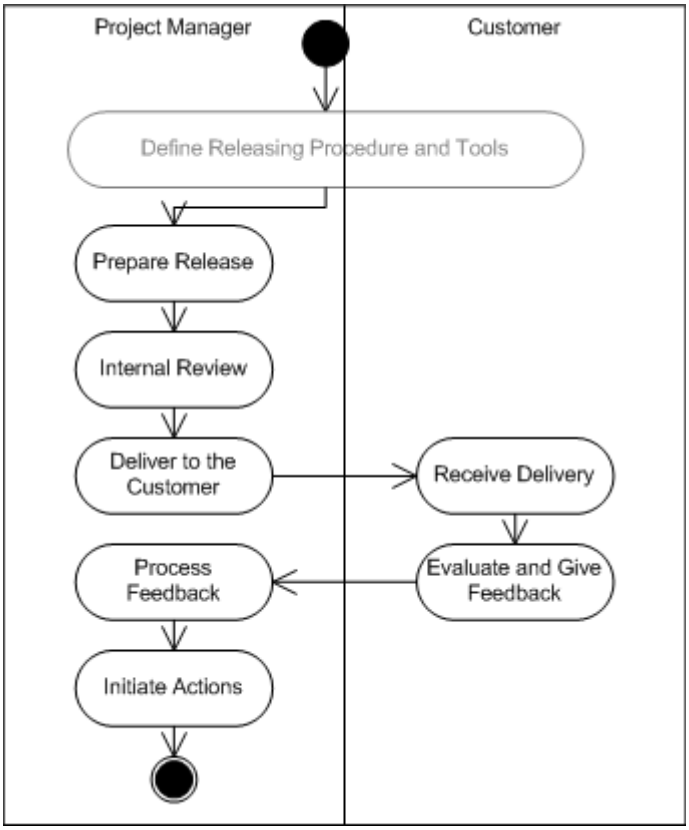

**Problem** The delivery of (intermediate) work results between the customer and the supplier has not been organized or is not working well.

**Initial Context** A new project has been started and Shared Understanding (KSP05) exists between the customer and the supplier about the expected results.

**Roles** *A Project Manager* representing the project and *a Customer Contact Person* accepting the work results.

**Forces** Based on the shared understanding of what the work results from the project shall be, the ways of delivering the work results must be defined, for example, by using certain configuration management tools.

**Solution**




1. *Define* together with the customer the *releasing procedure and tools*. Suggest the organization's standard if applicable. It is adequate to define this once if there are not bigger changes during the project affecting this. Based on what has been agreed on with the customer, plan internally, how the releases are implemented. Look also at possibilities to automate this procedure (including required testing) and initiate actions for that.
2. *Prepare releases*. Release can here mean from one to many artifacts planned to be delivered to the customer. It can be a large software release or a single document. Prepare the releases according to the procedure.
3. *Internal review.* Internally accept releases and initiate transfer of those to the customer.
4. *Deliver to the customer* and *Receive Delivery.*
5. The customer *evaluates* the release and *gives feedback*. In the case of official delivery, the feedback shall include approval (or complaint).
6. *Process feedback.*
7. *Initiate actions* if any additional actions are required based on the feedback.

In case the customer also delivers something to the supplier, this procedure should work also for that. Just to change the role names.

**Resulting Context** A project where the customer and the organization effectively share work-result knowledge between each other.

**Instances** The procedure definition to be implemented at least once for every project. During project realization the releasing to be implemented once per each relevant (document and/or software) release.

One potential pitfall is that releasing needs to be made often, but the procedure required for it is too massive and too time consuming.

**Process Connection** Project Management.

## Justification

**Basic Idea** Defining and implementing releases and releasing procedures between the customer and the supplier.

**Positive Instance** The pattern has been introduced according to the project practices in the studied organization. Release management (e.g. Sommerville, 2006, pp. 702-705) includes similar practices.

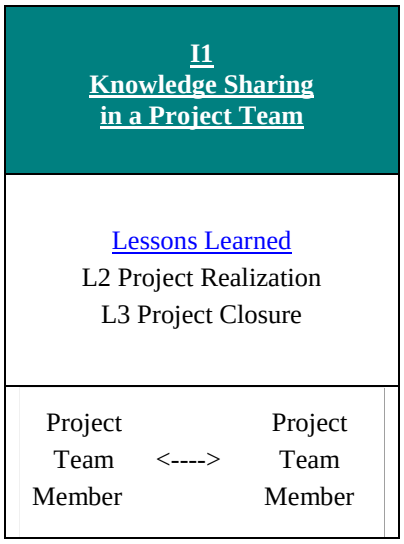
**Negative Instance** The releasing in project Alpha took too much time because of improper preparations and not automating the procedure. Better planning and actions based on that might have helped the project.

## References

Sommerville, I. (2006). Software Engineering. 8th Edition. Pearson Education.

## 15.17 KSP15 Discovered Lessons

Dimensions and Knowledge Flow:




**Problem** In a project many experiences are gained but those are not systematically collected and understood.

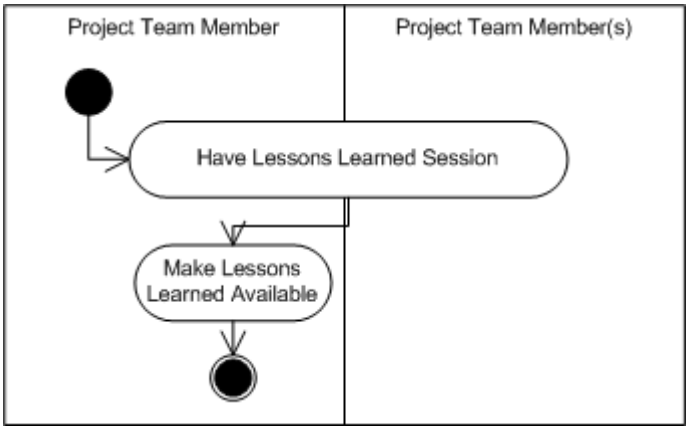
**Initial Context** Ongoing project or nearly closed project. At least some intermediate outcome is achieved.

**Roles** *Project Team Members* as a team of people working together.

**Forces** Most of the experience in a project business organization is gained in projects. To learn from the experience requires a translation from experience into knowledge, which takes a certain amount of intention (Dixon, 2000, p. 18). Thus, to know what has been learned and to make it possible later to share the knowledge, a project must be carefully analyzed. This also helps the project team members to understand what they have learned and to share this in their project team.

A team needs to explore the relationships between action and outcome in a project. The discussion related to this is what translates their experience into knowledge. A result of this exploring is the common knowledge in the team. Constructing team knowledge does not happen automatically. (Dixon, 2000, pp. 18-19).

**Solution**




1. *Have Lessons Learned Session*. In project team, explore and discuss the relationship between the action and the outcome. Think, for example:
    - How to value the outcome based on the effort taken?
    - How to value the outcome compared to the customer need and requirements?
    - What successes and problems have there been? Problems to be shared with others, potential best practices to be shared with others?
    - How the risk management worked and were there bigger problems that were not identified as risks?
    - Available competences compared to required competences?
    - Check also possible metrics and customer satisfaction information if available.
2. *Make Lessons Learned Available*. Document most important lessons learned. Notice especially lessons that could benefit people outside the team. See also Contributed Experience Base, KSP27.

**Resulting Context** A project systematically analyzed from the perspective of lessons learned. Organizational process for packaging and sharing knowledge initiated.

**Instances** Have lessons learned session at least for each relevant milestone. The meeting can be part of the milestone review or a separate meeting. A final lessons learned session should be taken during the project closure to summarize everything and (latest at this point) to initiate the organizational sharing of lessons learned (see: Contributed Experience Base, KSP27).

In addition to project teams, also other teams in an organization could think about having systematic lessons learned sessions.

One potential pitfall is that the lessons learned sessions are taken, but those start to be just some quick look through of measuring data and not having a real discussion that is required to translate the experience into knowledge.

**Process Connection** Project management: milestones review and project closure meeting.

**Related Patterns** Time for Reflection
Manns, M.L. and Rising, L. (2005). Fearless Change: Patterns for Introducing New Ideas. Pearson Education Inc., Addison-Wesley, 240-242.

## Justification

**Basic Idea** To have a team exploring the relationship between action and outcome and producing lessons learned based on that.

**Positive Instance** Creating common knowledge in a team, Dixon (2000, pp. 18-19).

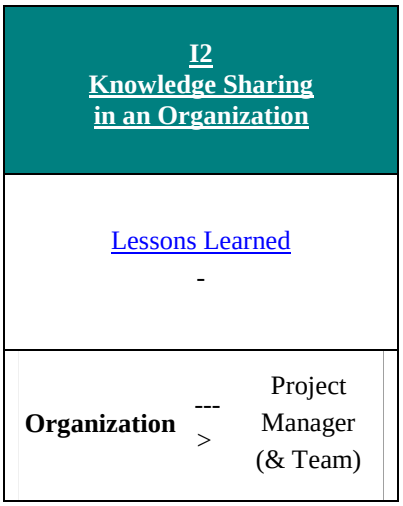
**Negative Instance** In project Alpha there were several mistakes made because of the loose connections to the other organization and to the other project teams. For example, better connections could have resulted in the enforcement of stricter process discipline and ultimately to a much more systematical requirements management.

## References

Dixon, N.M. (2000). Common Knowledge: How Companies Thrive by Sharing What They Know. Harvard Business School Press, Boston, Massachusetts.

## 15.18 KSP16 Established Experience Base

Dimensions and Knowledge Flow:




**Problem** Lessons learned are collected in projects but the organization does not have any systematic way to store and share those to support the work in projects.

**Initial Context** An organization not having a systematic procedure and environment to collect and share experiences (lessons learned).

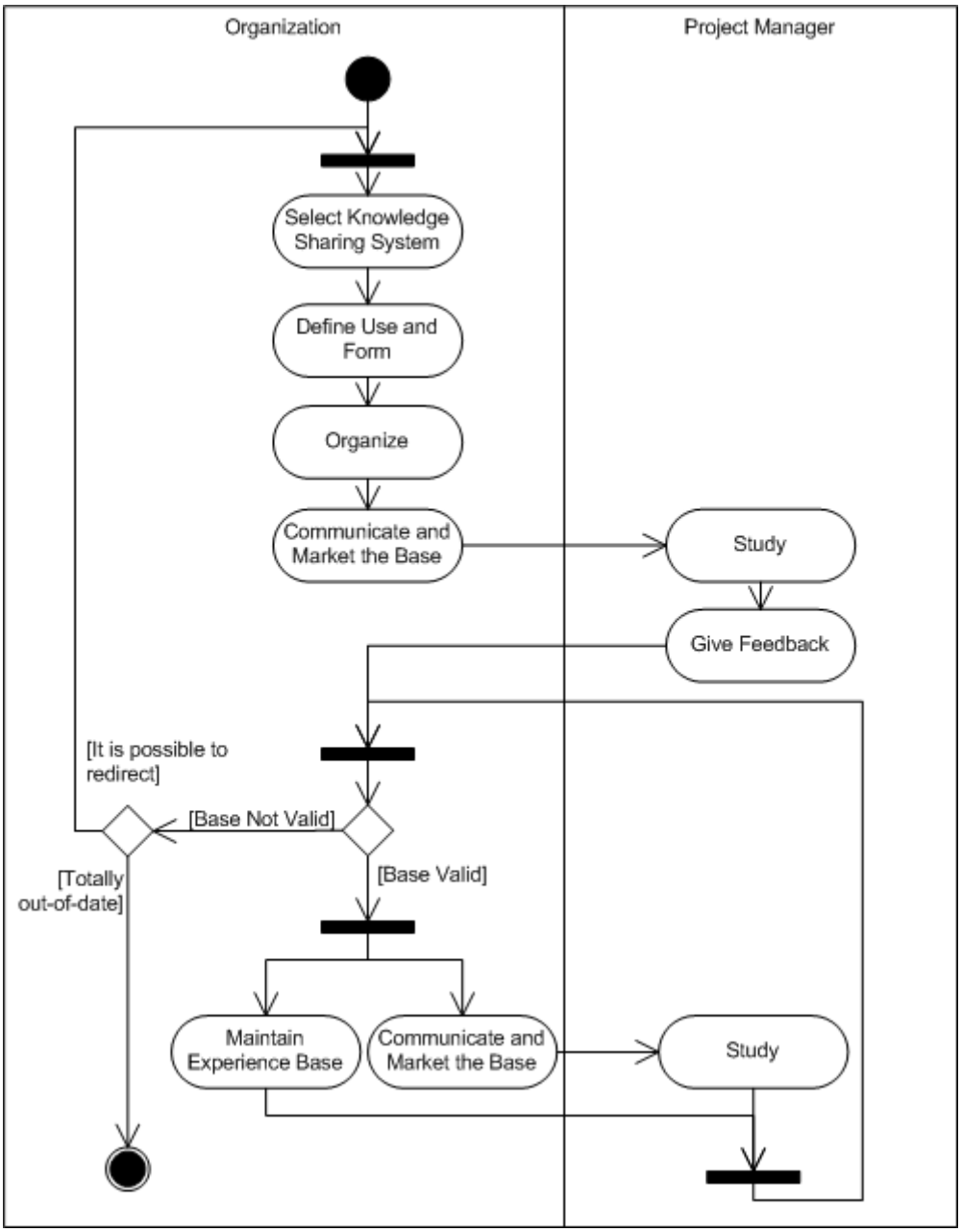
**Roles** An *Organization* represented by competence or quality management, for example. *Project Manager* (including other project team members).

**Forces** In organizations gained knowledge is "wasted" if not shared. The existing knowledge should not be difficult to find and reuse for project managers. (Komi-Sirviö et al. 2002.)

Many organizations start their knowledge sharing project by establishing a new database and just assuming, that people will use it when it is ready. But neither contributions nor retrievals occur with much enthusiasm. (Dixon, 2000, p. 2.)

Sharing knowledge takes time and effort from e.g. new knowledge development, but on the other hand, exploiting existing knowledge produces enormous cost savings (Dixon, 2000, pp. 19-20).

**Solution**




1. *Select Knowledge Sharing System.* Find a method for transferring knowledge to a group or individual that can reuse it (Dixon, 2000, p. 21). Concentrate especially on knowledge that should be created in projects and shared to other project teams (see Discovered Lessons, KSP15). Most probably one solution is an experience base, database for storing and sharing lessons learned, but remember also other ways of sharing knowledge and especially for supporting the use of the experience base.

2. *Define Use and Form.* Define what kinds of a form of knowledge could be useful with the selected methods. Define and pilot that. Have some example knowledge ready so it is easier for people to contribute.

3. *Organize.* Define owner for e.g. a resulting experience base. Define resources to maintain it.

4. *Communicate and Market the Base.* Communicate the existence of the experience base, start motivating people to contribute for it.

5. *Study* and *Give Feedback.* Study the experience base (etc.) and its possibilities. Give feedback about it and your needs for the knowledge to be shared.

6. *Maintain Experience Base.* Maintain the experience base according to the feedback from projects. Continue maintenance as long as the experience base is valid for the users. See also Contributed Experience Base (KSP27).

7. *Communicate and Market the Base* and *Study.* Support the continuous use of the experience base with communication and marketing activities in the organization. Continue this as long as the experience base is valid for the users.

If the experience base is not valid, but through reasonable actions it could be redirected to support the work, a revision and reinitiation could be made.

**Resulting Context** An organization having initiated an environment for systematic experience sharing.

**Instances** Establish once, maintain and support with communication continuously.

One potential pitfall is that the project managers and project team members will not see the benefits of this knowledge sharing system and thus not have the motivation to use it. The benefit of the system in practice is thus the most important criterion for the validity of the system.

**Process Connection** Supports process improvement and organizational development. This can have an affect on any of the processes.

## Justification

**Basic Idea** To systematically collect, store and share lessons learned.

**Positive Instance** Leveraging common knowledge in an organization, Dixon (2000, pp. 19-21).

**Negative Instance** Projects can be run without systematical lessons learned collecting and sharing, especially if an organization is very small and the lessons learned are shared as part of everyday collaboration. In a bigger organization, however, the same problems may occur again and again without having systematical experience / lessons learned sharing. In such a case, experience sharing is mostly based on unofficial knowledge sharing, for example, through a person's own private network (see My Network, KSP21).

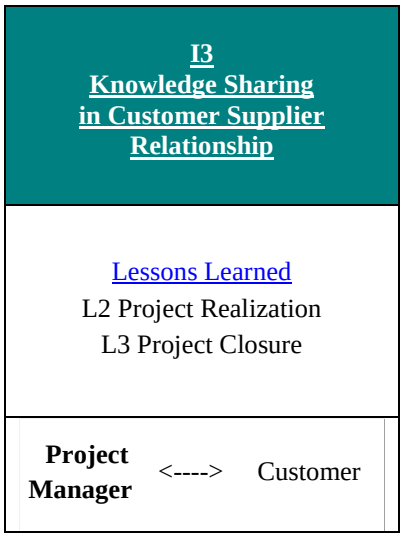
## References

Dixon, N.M. (2000). Common Knowledge: How Companies Thrive by Sharing What They Know. Harvard Business School Press, Boston, Massachusetts.

Komi-Sirviö, S., Mäntyniemi, A. and Seppänen, V. (2002). Toward a Practical Solution for Capturing Knowledge for Software Projects. IEEE Software, vol. 19, no. 3.

## 15.19 KSP17 Satisfied Customer

Dimensions and Knowledge Flow:




**Problem** The project manager does not know how the customer perceives the project and its results.

**Initial Context** Ongoing project or a project very recently closed. An organization having defined some formal way to ask for customer satisfaction information.

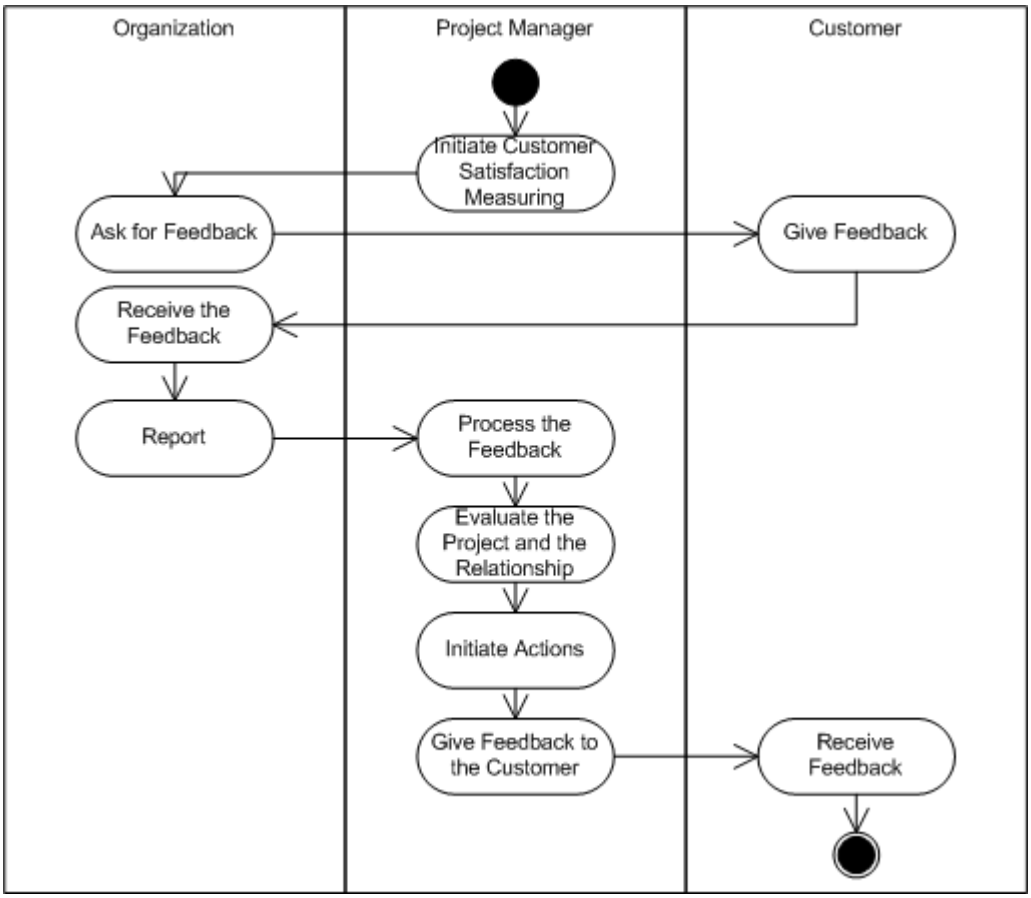
**Roles** *A Project Manager* and *a Customer Representative*. Also the organization is here present in the form of facilitating objective customer-satisfaction data collection.

**Forces** The success of a customer supplier relationship is based on the satisfaction of both parties. From a supplier perspective, it means especially, customer satisfaction. A good level of customer satisfaction may result in new projects with the customer and thus new possibilities for business for the supplier.

Customer satisfaction should be collected all along the way and actions should be initiated based on it. In addition some more formal way of collecting customer satisfaction information could be reasonable. To motivate the customer to give feedback, normally some actions are required. The best way to prove the usefulness of this is to have the customer informed about the results and to show that the results have really affected the way of working.

People in a project can initiate measuring, but they should not be the direct receiver of the results from the customer. If the customer would need to answer directly, for example, to the Project Manager, in some cases, some part of the feedback might be left out because the receiver has participated in the evaluated project.

**Solution**




1. *Initiate Customer Satisfaction Measuring*. A Project Manager initiates the customer satisfaction measuring.

2. *Ask for Feedback* and *Give Feedback*. Feedback is requested from a customer in an interview or by using a web survey.

3. *Receive the feedback* and *Report.* An organization receives the feedback and reports it or a summary of it to the project manager.

4. *Process the feedback.* Project Manager together with the project team processes the results. They plan possible required actions.

5. *Evaluate the Project and the Relationship.* Based on the customer satisfaction results and feedback from the team, the relationship and the whole project is evaluated. What have worked well and what should be improved? This could be part of a lessons learned session (see Discovered Lessons, KSP15). The resulting lessons learned shall be shared with the organization (see Contributed Experience Base, KSP27).

6. *Initiate actions*. If required, initiate corrective and preventive actions to improve the relationship and performance.

7. *Give Feedback to the Customer* and *Receive Feedback.* Inform the customer at the general level about the results and show in practice that actions have been taken to improve the relationship.

**Resulting Context** Well working, open customer supplier relationship with parties committed to further improve the relationship.

**Instances** Utilize this pattern at least once per each relevant project. Use it preferably before closure, when the customer starts to see the outcome of the project, but when there is still some time to do possible required corrective actions during the project.

One potential pitfall is that the customer is not willing to give feedback. Then new ways of asking for it and analyzing it are required.

**Process Connection** Customer satisfaction follow-up.

**Related Patterns** Engage Customers

http://users.rcn.com/jcoplien/Patterns/Process/section21.html

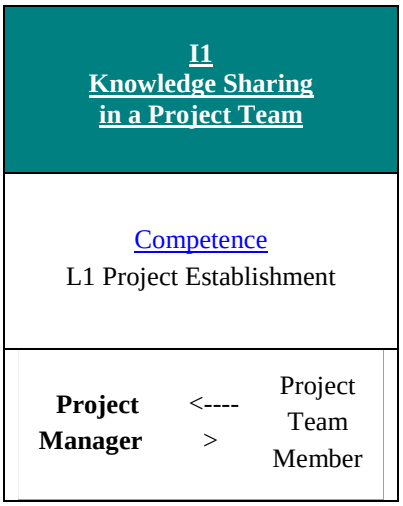
## Justification

**Basic Idea** To learn from the customer supplier relationship, systematic evaluation of the relationship and customer satisfaction is required.

**Positive Instance** The pattern follows the systematically utilized customer satisfaction follow-up procedure at the organization being studied. Practices used by several projects in the organization.

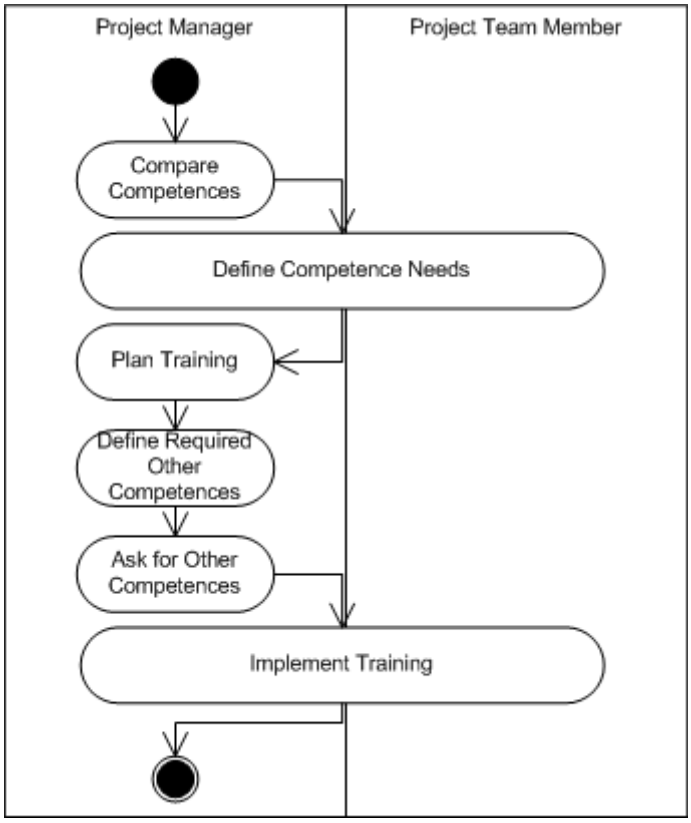
**Negative Instance** Not valid here. A project can be implemented without formal customer satisfaction follow-up. Utilizing a systematical approach from this aspect would result in a better understanding how to improve the customer supplier relationship and through that to create more business.

## 15.20 KSP18 Improved Competences




| | |
|---|---|
| **Problem** | A project manager does not know how well the competences of the project team match the competence requirements of the project. |
| **Initial Context** | A new project is starting and the organization has named a project team to implement it. |
| **Roles** | A *Project Manager* with the *Project Team Members*. |
| **Forces** | To make a project efficiently the required competences should be available either in the project team or externally. Partially, the competences can be trained/learned during the project.<br>In some projects, it is not possible to have all required competences available and known when starting. For example, projects dealing with new technologies with which no one are yet familiar. |

**Solution**




1. *Compare Competences.* Compare available competences to the required ones. See Assured Resources (KSP19) for defining required competences.
2. *Define Competence Needs.* What competence needs are not covered by the assigned project team? What additional competences are required? Define those together with the project team if possible.
3. *Plan Training*. If some competences can be gained through training during the project, plan training.
4. *Define Required Other Competences*. Define what competences are not covered in the project team and those that can not be acquired through training by the project team and need to be available some other way.
5. *Ask for other Competences*. Ask from the organization, see Assigned Experts, KSP03. Examples of other resources could be internal experts or external resources. If required competences can not be made available, follow and manage this as a risk item.
6. *Implement* required *training*.

**Resulting Context** A project either with a good access to required competences or a good understanding of missing competences (e.g. new technology) and follow-up of those as an risk item.

**Instances** Use this pattern always when establishing a new project.

One potential pitfall is that the identification of the competence requirements is not correct and then the competence need definition and actions might also fail.

**Process Connection** Project Management Process - Project Establishment

**Related Patterns**

- Domain Expertise in Roles
  http://users.rcn.com/jcoplien/Patterns/Process/section7.html
- Apprenticeship
  Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 108-109.
- Day Care
  Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 88-91.
- Informal Learning Groups
  http://www.publicsphereproject.org/patterns/print-pattern.php?begin=98

## Justification

**Basic Idea**

Availability of required competences needs to be ensured either in the project team or through external support to the project team.

**Positive Instance**

This practice is a normal part of project planning in the organization which is being studied. This practice is based on the idea of expert transfer (Dixon 2000, pp. 127-140). If a project does not have all the competences required right in the beginning they can be made available to the project by using experts who have those required competences. The experts can be used either to train the project team or to act as experts when needed.

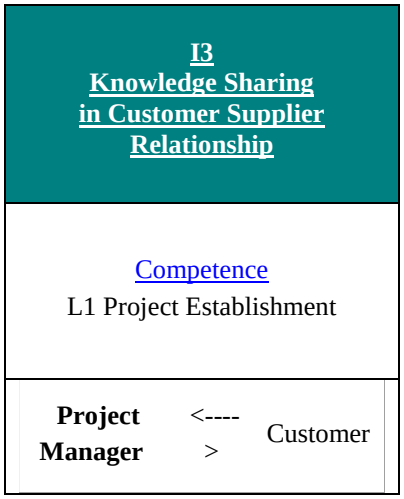
**Negative Instance**

Project Alpha did not have adequate competences in the project team; neither was there any external support arranged for before the problems started to be visible outside the project team. This pattern could have helped to evaluate the situation better. It is not sure, however, if that would have happened because when the project started there were different levels of understanding of the requirements between the supplier and the customer. That difference also affected the understanding of the competency requirements.

## References

Dixon, N.M. (2000). Common Knowledge: How Companies Thrive by Sharing What They Know. Harvard Business School Press, Boston, Massachusetts.

## 15.21 KSP19 Assured Resources




**Problem** An organization not being sure if it has available adequate resources for a new project.

**Initial Context** A customer has requested a new project.

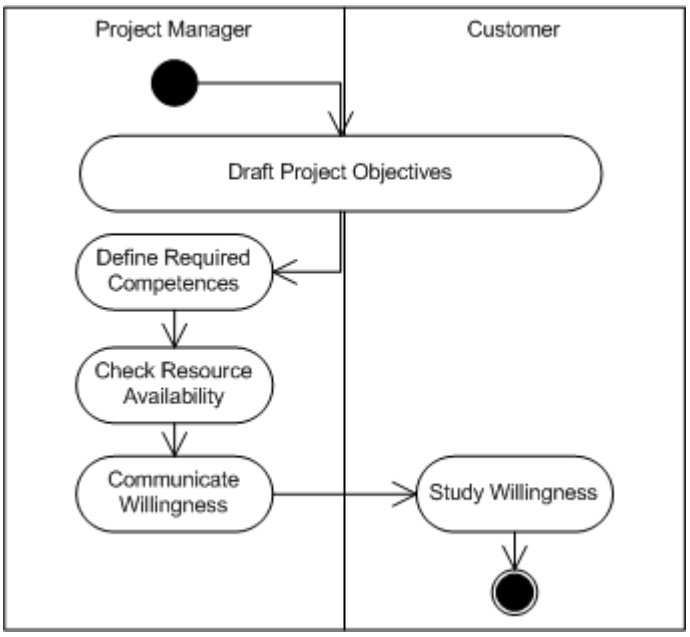
**Roles** *A Project Manager* together with *a Line Manager* having the resource ownership. One or more *Customer Representatives* telling their expectations and needs. Also sales is normally cooperating with a Project Manager in this.

**Forces** To commit to a project, at least the most critical resources need to be known and pre-assigned. Resource refers here to human resources. It includes the quality, the competence areas and levels, and the quantity--the amount of required resources.

To have some certain competence area and level available does not always mean that it should be possessed by the current employees. Also suppliers and recruitments can be thought as a solution.

Especially, when the sales and project delivery organization are separate organizations this need for checking available resources requires extra effort but is very critical to the successful start of a project.

**Solution**




At the project level the following actions are required:

1. *Draft Project Objectives* together with the customer. It could be an early version of the Shared Understanding (KSP5).

2. *Define Required Competences* based on the draft objectives, define required competences. For example, what technology areas are involved, what level of competence (basic, average, high) is required and how many persons would be required? Check also the match with the vision and strategy.

3. *Check Resource Availability*. Check that it would be possible to resource this kind of a project. The resources can be from the own resources or supplier resources.

4. *Communicate (and Study) Willingness*. If the resources could be available for a project, communicate willingness to the customer. If resources can not be confirmed on an adequate level, communicate to the customer that the project is not possible in the current situation.

**Resulting Context** A project having required competences adequately identified and required resources assured for the project.

**Instances** Implement this always when planning a new project without the resources confirmed.

**Process Connection** Project management, resource management.

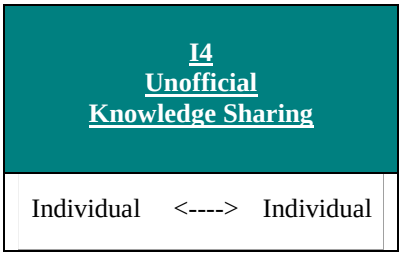
## Justification

**Basic Idea** Pre-assigning required resources for a new project to ensure their availability.

**Positive Instance** A practice observed in several projects in the organization being studied. Successful implementation of this especially included a proper check for resource availability that would be supported with open internal communication between project manager and sales and line management about different resourcing possibilities.

| | |
|---|---|
| **Negative Instance** | This includes projects where the sales department has sold a project that the delivery organization does not know how to implement because of lack of available resources. |

## 15.22 KSP20 Initiated Communication




**Problem** Communication structures of an organization need to be strengthened.

**Initial Context** An organization willing to support the emergence of positive unofficial networking.

**Roles** *Individuals* at the organization.

**Forces** Expertise is deeply embodied knowledge and requires human-to-human processes through which this expertise is triggered and shared (Fitzpatrick, 2003). Also, like Haas et al. (2003) notice, "communication, both formal and informal, is the main driver of success in a community of practice".

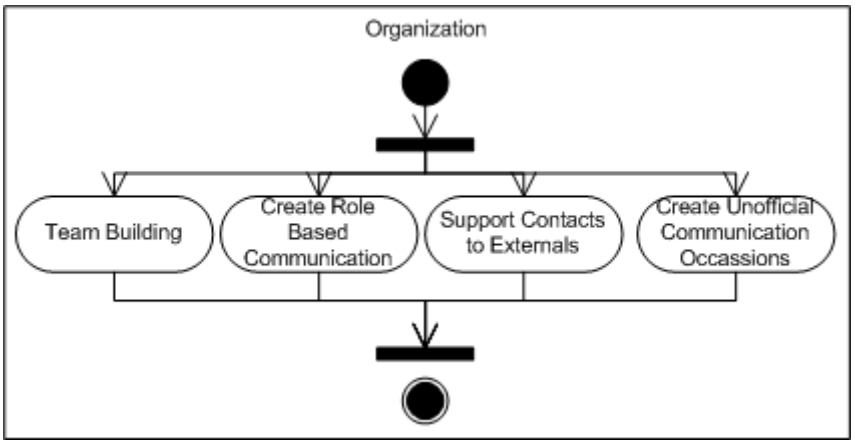
Haas et al. (2003) report about the organization of Chrysler. Earlier they had a stovepipe organization, where design passed the work on to engineering and so on. This resulted in insufficient communication and collaboration between the different functions and costly re-do loops occurred. To overcome this they changed to a platform-based model bringing together all development stakeholders of a vehicle in teams. After a while the platforms started to be a sort of lateral stovepipes and, again, lack of communication was the fact, now between the teams for optimized production in the original functions.

If a project team works well together also the informal communication is most probably more efficient and supports the project work. Team building activities will support the familiarizing to the team and to the people there.

In addition to good communication in a project team, communication between project managers or between architects, software engineers etc. is important especially from the perspective of improving the work practices.

An organization, represented by the management, can not force people to communicate, but an organization can make it possible for people to meet each other, to know each other and to try to initiate communication and collaboration in reasonable sub-groups.

**Solution**




To initiate unofficial communication it is required that people know each other (for example: prepare more formal situations where they are introduced) but also making possible unofficial situations to meet each other. The first three introduced actions are more formal and the fourth is to make possible the unofficial situations.

1. *Team Building*. With team building exercises it is possible to build strong core teams to be basis for project teams.

2. *Create Role Based Communication*. Examples of this could be like monthly or so project manager forum, internal software engineering or technology days. Also, officially announcing experts (see Named Experts, KSP02) could be a start for this. Remember also cross-role type of communication. This can create new kind of ideas.

3. *Support Contacts to Externals*. Externals can provide new knowledge for the organization. In practice this could mean e.g. sending personnel to seminars, fairs etc. or having externals as consultants etc. working in the organization. The limits of information security, however, shall be clear to everyone communicating with externals.

4. *Create Unofficial Communication Occasions*. In addition to formally making possible people meeting each other, free space is required for them to continue communication. For this, arrange spaces in the office for water coolers, coffee rooms, entertainment etc. Also, arrange entertainment situations for the personnel to make it possible for them to see each other freely. This would be especially important if an organization has several offices and people see each other very seldom.

The purpose of this pattern is to create possibilities for knowledge sharing to take place during and especially after these steps in the form of unofficial knowledge sharing. Pattern My Network (KSP21) looks at this same thing from the perspective of an individual.

**Resulting Context** An organization that has possibilities for strong multidimensional knowledge sharing including unofficial knowledge sharing.

**Instances** Always when an organization feels that the communication networks should be strengthened. Most of the activities introduced are continuous activities that should be thought to be on-going.

One potential pitfall is that much unofficial knowledge sharing is created but it is not positive to the organization.

**Process Connection** No direct reference to processes.

**Related Patterns**

- Responsibilities Engate
  Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 209-212.
- The Watercooler
  Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 226-228.
- Matron Role
  Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 140-141.
- Do Food
  Manns, M.L. and Rising, L. (2005). Fearless Change: Patterns for Introducing New Ideas. Pearson Education Inc., Addison-Wesley, 132-134.
- Informal Learning Groups
  http://www.publicsphereproject.org/patterns/print-pattern.php?begin=98
- Thinking Communities
  http://www.publicsphereproject.org/patterns/print-pattern.php?begin=118

## Justification

**Basic Idea**

To give possibilities for personnel to initiate their own communication networks.

**Positive Instance**

"To close strategic gaps in knowledge flow, informal communities of engineers who formerly worked together in the stovepipe organization but who were separated because of the platform reorganization started to appear. Their initial agenda was an informal exchange of best practices and lessons learned at the different platforms...Management immediately recognized the importance of these communities to ensure a two-dimensional matrix structure for knowledge flow." Haas et al. (2003, pp. 181-182).

The platform team could be compared to a project team requiring role-based communication and collaboration over projects. In addition to this two-dimensional communication, also the externals and free internal communication dimension has been added.

**Negative Instance**

If, in project Alpha, the newcomer project team members would have been a more effective part of the existing unofficial networks inside the organization they would have received much more support through those networks This might have helped to avoid some of the problems.

## References

Fitzpatrick, G. (2003). Emergent Expertise Sharing in a New Community. In book Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise: Beyond Knowledge Management. MIT Press.

Haas, R., Aulbur, W. and Thakar, S. (2003). Enabling Communities of Practice at EADS. In book Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise: Beyond Knowledge Management. MIT Press.

## 15.23 KSP21 My Network

| I4<br>Unofficial<br>Knowledge Sharing |
| --- |
| Individual <----> Individual |

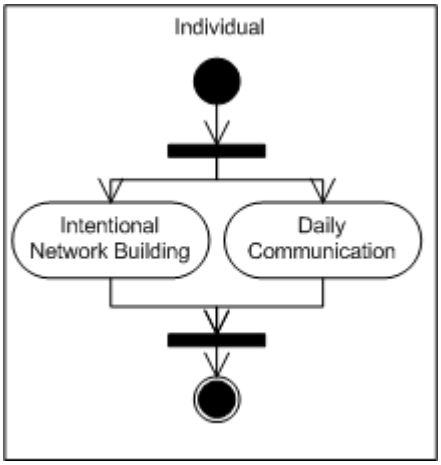
**Problem** Not strong enough personal communication network.

**Initial Context** An individual wanting to establish or to strengthen a personal communication network.

**Roles** *Individuals* at the organization.

**Forces** To succeed in work and life an individual must have a network of people with which she/he can communicate and collaborate. It can include friends, colleagues etc. The network can be built intentionally or it can start to exist through normal participation in social communication.

**Solution**




1. *Intentional network Building* could include actions like:
    - Systematically defining what kind of persons would be important in the network and initiating contacts with them. Remember, however, that you can not force others to have you as part of their communication network.
    - Joining associations and communities with similar interests to your own.
    - Utilize all possibilities to meet new people. Introduce yourself actively to other people.
    - Define key persons in your organization and other important organizations. Contact them in some suitable situation

2. *Daily communication*. Be active in everyday situations. Introduce yourself to others and be active in communication. Be open to friendships and build trusting relationships with persons you like. Remember that you also need to give something in order to get friendship and/or good cooperation.

**Resulting Context** An individual having initiated strong communication network.

**Instances** Always when an individual feels that the communication networks should be strengthened. The daily communication is a continuous action.

One potential pitfall is that a person is building a network, but not taking enough time and activities to have it as an active network. A passive network is more like a list of names without people really wanting to help you.

**Process Connection** No links to organizational processes. Part of individual processes.

**Related Patterns** Stay in Touch
Manns, M.L. and Rising, L. (2005). Fearless Change: Patterns for Introducing New Ideas. Pearson Education Inc., Addison-Wesley, 221-223.

## Justification

**Basic Idea** To initiate and strengthen a personal communication network.

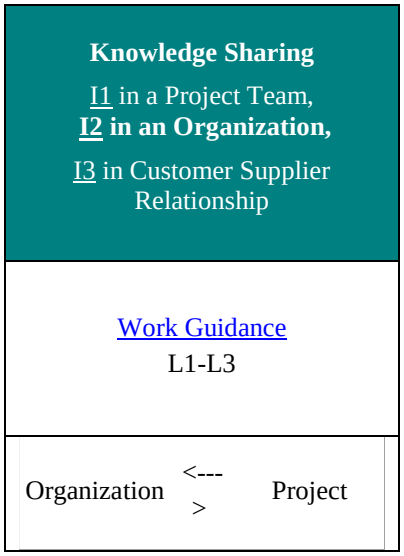
**Positive Instance** This has been observed by many individuals. Within the Internet there exists several communities that support the creation of their own personal networks (For example, LinkedIn , www.linkedin.com). Also (Ehrlich 2003) highlights the importance of building up a set of personal connections that can be called upon at short notice for joint problem solving.

**Negative Instance** People not having personal contacts with others can not get strong support from their own personal contact network.

## References

Ehrlich, K. (2003). Locating Expertise: Design Issues for an Expertise Locator System. In book Ackerman, M., Pipek, V. and Wulf, V. (Eds.) Sharing Expertise: Beyond Knowledge Management. MIT Press.

## 15.24 KSP22 Work Guidance



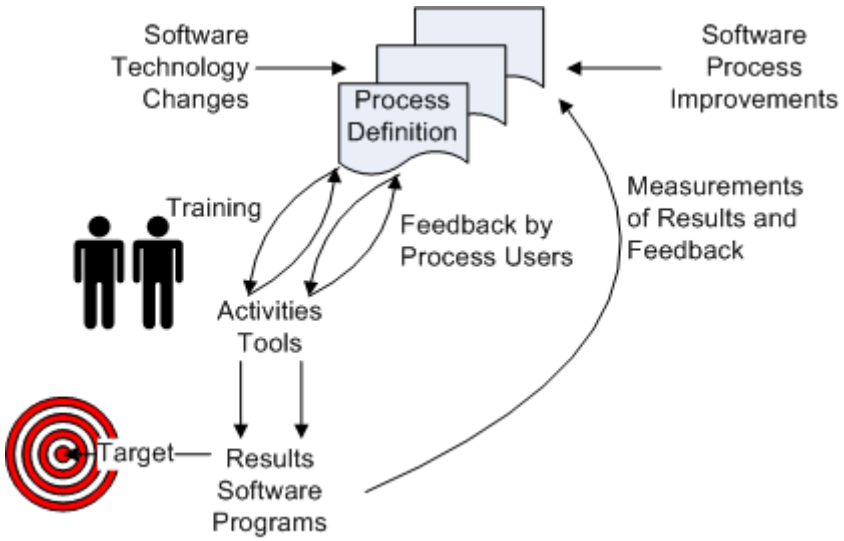




Figure 1. Effective software process environment (modified from Zahran, 1998, p. 68).

**Problem** An organization with a need to establish or improve the guidance of work in the organization in order to allow for more efficient team work.

**Initial Context** An organization in the software engineering business that has initiated process management. At the project level Shared Understanding (KSP05) exists between the customer and the organization.

**Roles** An *Organization* represented, for example, by top management and quality management. A project represented by all *Project Team Members*.

**Forces**

To guide work in an organization, a communicated and shared target of work is required as well as common processes about how the work should be done. The target guides by showing where to go and the common processes that facilitate team work to achieve the goals.

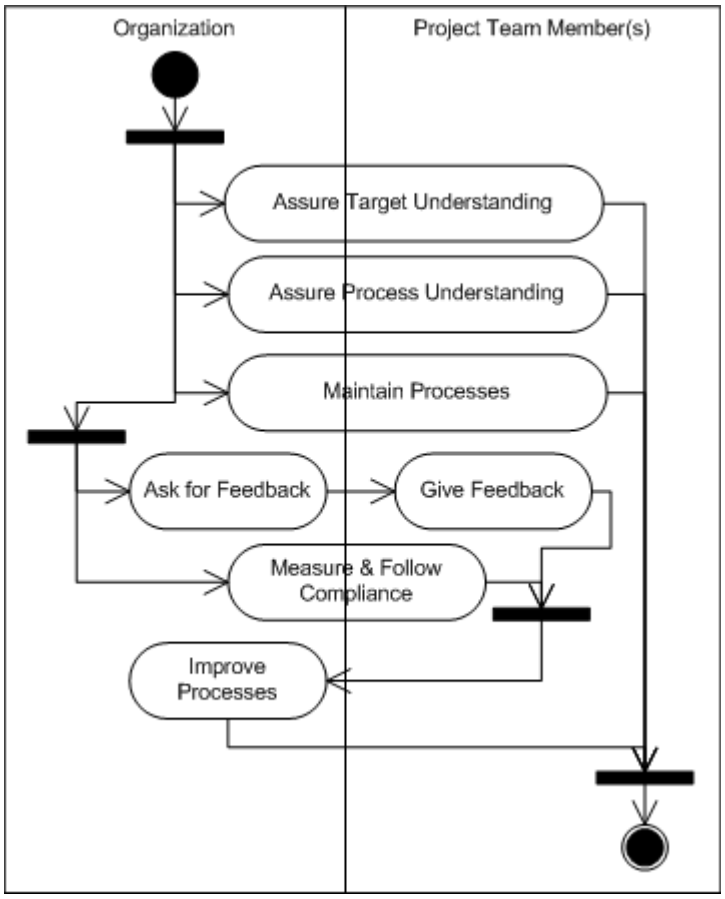
Aspects of a process (Zahran, 1998, p. 6-7):

- Process is the document specifying the process.
- Process is knowledge in people's brains to drive their behavior.
- Process is in the results of the process activities.

According to Zahran (1998, p. 41, see also the Figure 1) the following factors increase process effectiveness:

- Living process documentation having dedicated resources for owning, disseminating and maintaining it.
- Everyone in the organization is keen to follow the process.
- Feedback is gathered, for example, through performance measurements and utilized in process improvement.
- The process impact on the business goals is evident and following the process is the norm.

**Solution**




1. *Assure Target Understanding.* Assure that at all levels of the organization the target of the work and business is clear. Assure also, that the process improvement work supports the implementation of the organization's vision

and strategy.

2. *Assure Process Understanding.* Assure that people know and are adequately trained to use the organization's set of standard processes.
3. *Maintain Processes.* Continuously maintain the processes. Check also how the environment, e.g. software technologies change and define how those affect the processes.
4. *Ask for Feedback* and *Give Feedback.*Constantly *c*ollect process user feedback.
5. *Measure & Follow Compliance.* Constantly collect relevant measurement data and information about the compliance to processes and their efficiency.
6. *Improve Processes.* Based on the organization's targets, and the results from steps 4 and 5, plan and implement process improvement activities.

This is a continuous procedure, so when the closing point has been reached it must be restarted.

**Resulting Context** An organization having well established process management and target understanding.

**Instances** To initiate the guidance in an organization and to maintain and improve it continuously.

One potential pitfall is to improve and maintain the processes fully separated from the real work resulting in processes that are not reasonable to follow in real life.

**Process Connection** Business Management and Process Management.

## Justification

**Basic Idea** To define common targets and common ways of working and continuously improving them.

**Positive Instance** The whole book of Zahran (1998) and e.g. Galin (2004, pp.65-66):
"Procedures and work instructions are based on the organization's accumulated experience and knowledge; as such, they contribute to the correct and effective performance of established technologies and routines. Because they reflect the organization's past experience, constant care should be taken to update and adjust those procedures and instructions to current technological, organizational, and other conditions."

**Negative Instance** In the beginning of project Alpha the project manager commented that processes make him" feel sick". During the project, however, all team members started to realize that they could have avoided many pitfalls if they would have utilized the organization's standard processes as their common way of working.

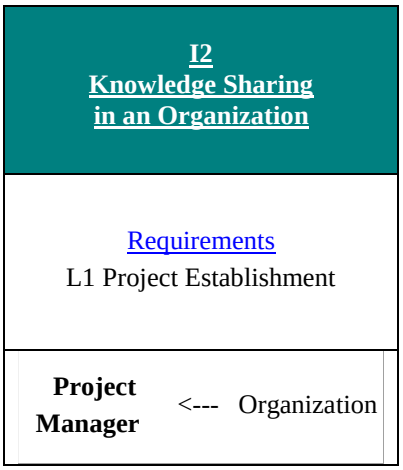
## References

Galin, D. (2004). Software Quality Assurance: From theory to implementation. Pearson Education, Addison-Wesley.

Zahran, S. (1998). Software Process Improvement: Practical Guidelines for Business Success. SEI Series in Software Engineering, Addison-Wesley.

## 15.25 KSP23 Not Wasted

Dimensions and Knowledge Flow:




**Problem** Requirements for a new project must be defined, but time available is very limited.

**Initial Context** A new project is planned or is starting and requirements need to be defined. Sets of Reference Requirements (KSP07) exist.

**Roles** *A Project Manager* or Requirements Manager defining requirements. *Organization* represented by a person being the named owner for the reference requirements sets.

**Forces**

Requirements for a project are normally defined under strict time pressure. If then, some earlier defined and tested sets of requirements could be used, it would shorten the required time period and also result in more quality requirements compared to the situation where all requirements have to be defined starting from scratch.

When using existing requirements the intellectual property rights (IPR) must be checked. Some projects might be such that the rights belong to the customers. Those requirements are not allowed to be used unless a permit has been obtained from the customer.

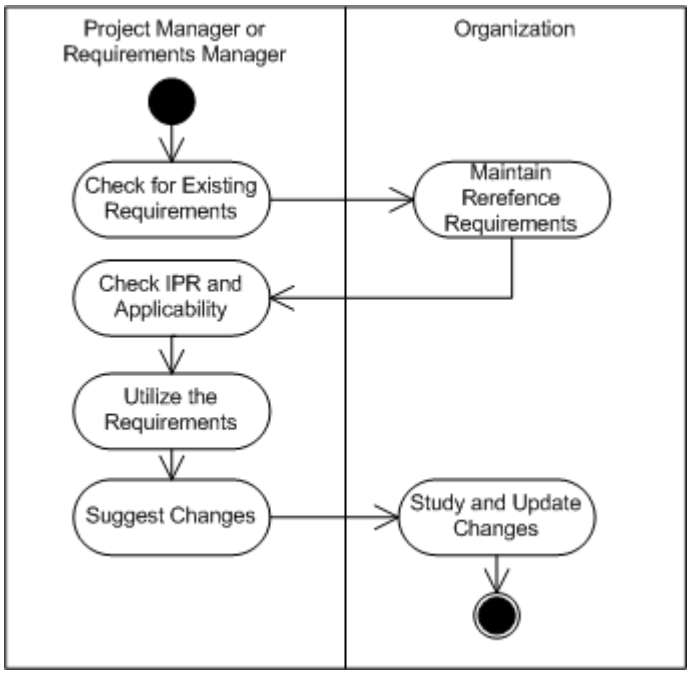
Using existing requirements or use case components will gain a number of benefits. Jacobson et al. (1997, pp. 117-118) have listed the following benefits from the perspective of reusing use case components:

- Effectiveness: using existing components as a toolbox expressing known system functionality -> faster and more accurately capturing of requirements.
- Quality: improving the quality of the requirements capture by reusing "certified" use case components and not missing some less obvious requirements when choosing from "complete" set of use cases.
- Predictability: improving project estimates through reusing use cases of which much is already known.

- Uniformity: reusing use case components ensures consistency of terminology and system.
- Design reuse: reusing use case components makes it possible to reuse the design and implementation of that use case.
- Rapid learning: the use cases provide usage-oriented documentation of the component system and thus the intended use of the component system can be seen by looking at the use cases.

Reuse of requirements can be direct, where a requirement from one system is taken as a requirement with minimal changes, or it can be indirect, where existing requirements are used to prompt users for their specific requirements (Sommerville & Sawyer 1997, pp. 106-107).

**Solution**




To use reference requirements:

1. *Check for Existing Requirements.* When defining requirements check for existing reference requirements: in the organization and external ones.
2. *Maintain Reference Requirements.* Support projects how to use the existing reference requirements. Study needs and decide maintenance activities to the reference requirements. (See also: Reference Requirements, KSP07)
3. *Check IPR and Applicability.* Check applicability of potential sets of reference requirements. Check the IPR situation (availability for the project, possible needs for protecting own IPRs).
4. *Utilize the Requirements.* Check the terms of use and use the requirements according to that. The requirements can be used as is or as a basis when defining the requirements with the customer (Discovered Bones, KSP06).
5. *Suggest changes*. If changes need to be implemented requirements, check if those would be useful also in the reference requirements. If yes, suggest those changes.
6. *Study and Update Changes*. Study suggested changes and decide possible

actions.

| | |
|---|---|
| **Resulting Context** | Quickly and effectively defined requirements and through earlier tested requirements improved quality compared to starting from scratch. |
| **Instances** | Pattern to be used always when defining requirements. Maintenance should be continuous during the lifetime of the reference requirements set.<br><br>One potential pitfall is to use a reference requirements set, that is not really applicable but needs a lot of tailoring. |
| **Process Connection** | Software engineering, requirements definition. |
| **Related Patterns** | Get On with It<br>Coplien J.O., Harrison, N.B. (2005). Organizational Patterns of Agile Software Development. Lucent Technologies, Pearson Prentice Hall, 38-41. |

## Other Practices

## Justification

| | |
|---|---|
| **Basic Idea** | To speed up the requirements definition through using earlier defined requirements. |
| **Positive Instance** | Reuse of requirements (Jacobson et al. 1997). |
| **Negative Instance** | Not valid here. A project also can be made without using any earlier defined requirements, but the amount of time to be used might be longer. If there will be similar projects / project results after the project, then it could be reasonable to consider if some part or all of the requirements could be defined and supported to be reused later. |

## References

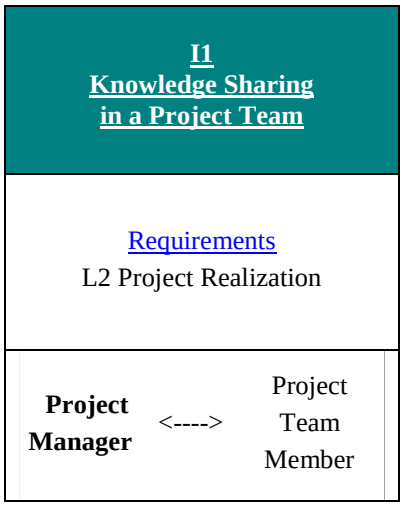
Clements, P. and Northrop, L. (2002). Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley.

Jacobson, I., Griss, M. and Jonsson, P. (1997). Software Reuse: Architecture, Process and Organization for Business Success. ACM Press, Addison-Wesley.

Sommerville, I and Sawyer, P. (1997). Requirements Engineering: A good practice guide. John Wiley & Sons, Chichester, UK.

## 15.26 KSP24 Flexible Skeleton

Dimensions and Knowledge Flow:




**Problem** During a project, requirements change and the project team members do not know what changes must be implemented and how those affect the project.

**Initial Context** A project is ongoing and Created Skeleton (KSP08) exists for it.

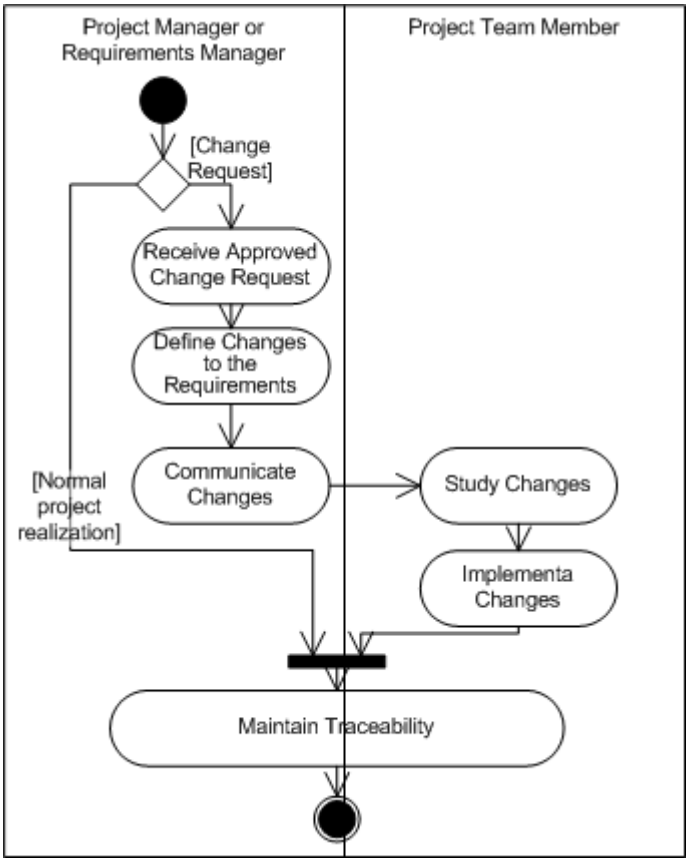
**Roles** *Project Team Members* and *a Project Manager* or a person in the role of Requirements Manager.

**Forces** Requirements are a structured way to introduce what is required from a project.

Requirements are linked to all parts of the project implementation through the traceability information, and thus are a sort of a skeleton for the project. The traceability information does not help unless it is kept up-to-date.

Changes to requirements must be systematically processed and the effects to other requirements, work results etc. studied. Changes must be properly communicated so that all project team members (and other relevant stakeholders) know them.

**Solution**




1. *Receive Approved Change Request* and *Define Changes to the Requirements*. Based on approved change request, define the changes to the requirements. Update the requirements specification.

2. *Communicate Changes*. Communicate the changes to the project team members. Notice especially the persons who will implement the changes, but also inform others at the general level. E.g. inform briefly in project team meetings.

3. *Study Changes* and *Implement Changes.* Study required change and what parts of the software it will affect. Utilize the traceability information for this. Implement the change.

4. *Maintain Traceability*. Maintain the traceability information.

This is defined here as a one-time procedure but, after completing, it a new round needs to be started as long as the project realization continues.

**Resulting Context**

Requirements and the linkage of those to the other parts of the project are well known in the project team making possible cooperation, separate sub teams working with separate tasks and quick understanding of how suggested and implemented changes affect the project.

One potential pitfall is not noticing all work results etc. being affected by a change. This could result in a situation where part of the project team continues building a module that will not work because of changes in some other module.

**Instances**

To be used continuously in projects. If a project does not contain many software requirements it still normally contains other project requirements (e.g. integration

projects).

**Process Connection** Requirements Management

## Justification

**Basic Idea** To have well working change management based on requirements utilizing traceability information.

**Positive Instance** Change management and bi-directional traceability introduced as special practices e.g. in CMMI (Chrissis et al., 2003, pp. 490-491) of requirements management process area. Steps defined here following that and the practice was observed in many projects in the studied organization.

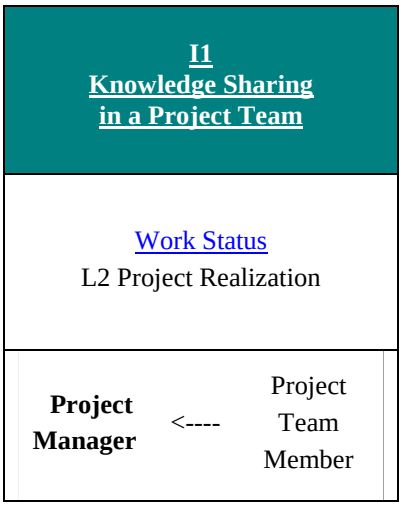
**Negative Instance** At project Alpha, there were many changes during the project. While not having well established requirements and change management, one of the many resulting problems was that for the distant project team members it was difficult to do the tasks when the others changed the environment continuously and did not properly communicate it.

## References

Chrissis, M.B., Konrad, M. and Shrum, S. (2003). CMMI - Guidelines for Process Integration and Product Improvement. SEI series in software engineering, Addison-Wesley.

## 15.27 KSP25 Followed Progress

Dimensions and Knowledge Flow:




**Problem** A project manager not knowing the project situation and progress status well enough.

**Initial Context** A project having a Scheduled Baseline (KSP09).

**Roles** *A Project Manager* managing the project and *Project Team Members* implementing the project.

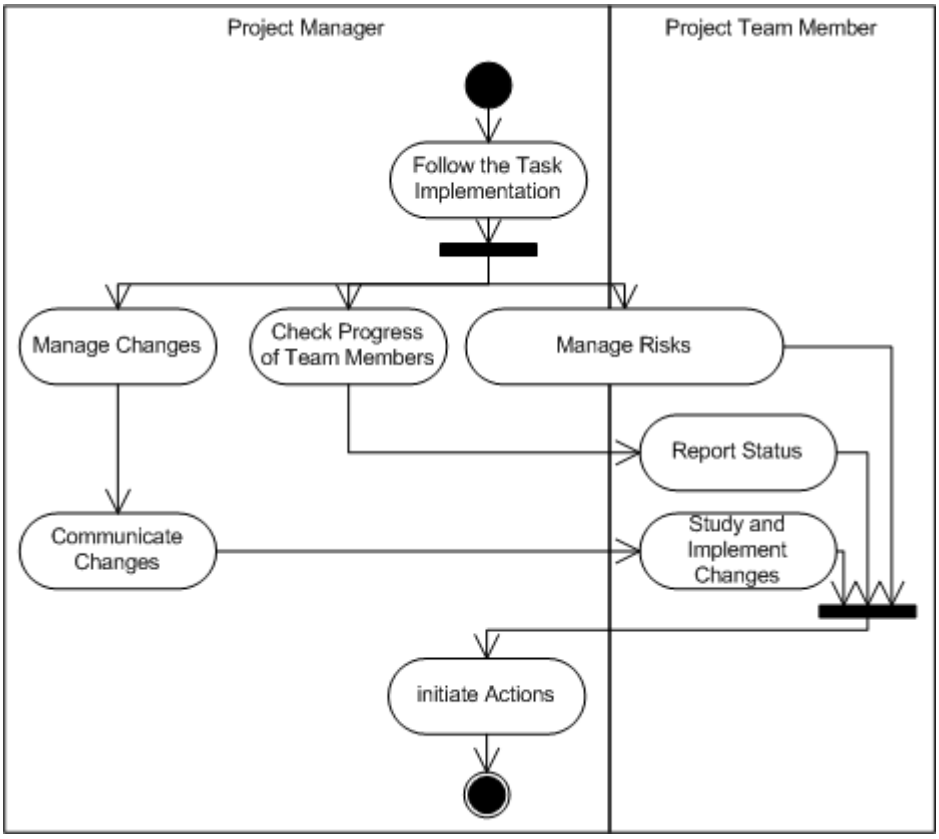
**Forces** Project Manager should constantly know what the real status of a project is compared to project objectives and plans. Without this knowledge, managing a project would be like driving in dense fog.

Well defined tasks create a basis for progress follow-up. If tasks are defined small enough but still have well defined results it is possible to follow progress through the realization of those results.

Changes to tasks need to be managed and communicated in a project team.

Risk identification allows preventive actions before the identified risk is realized.

**Solution**




To have adequate work status knowledge the project manager shall:

1. *Follow the task implementation*. Compare the implemented tasks to the project plan and schedule at the general level. When reasonable, utilize also additional methods, for example, critical path.
2. *Check Progress of Team Members & Report Status*. Follow the progress of team members (see also: Trust or Check, KSP04)
3. *Manage risks.* Identify and follow risks together with the team.
4. *Manage changes* to the plan and requirements (see also: Flexible Skeleton, KSP24). A change may affect also the project plan. If so, update the plan and have it approved.
5. *Communicate Changes* & *Study and Implement Changes.* Assure that the project team knows the required changes and their influence.
6. *Initiate Actions*. Based on the results of follow-up, initiate required preventive and corrective actions.

**Resulting Context**

A Project Manager running a project where the work status is adequately and continuously known. Risk management is implemented to avoid identified possible problems later.

**Instances**

This is a continuous activity during existing projects. The whole act should be implemented monthly/weekly/daily based on the intensity and criticality of a project.

One potential pitfall is that some engineer reports that the task progresses ok, but that is not the case in practice. See Trust or Check, KSP04 for avoiding such a pitfall.

**Process**

Project Management process, project execution.

**Connection**

## Justification

**Basic Idea**

To follow project progress on task basis and comparing the realization to the project plan and schedule.

**Positive Instance**

Follow-up is a basic action in a project to know the status of a project. This pattern can be found in most of the projects in the organization being studied and it is recorded into the processes of the organization.

Monitor Project Against Plan is also a specific goal in CMMI (Chrissis et al. 2003, pp. 392-398) as part of the project monitoring and control process area.

**Negative Instance**

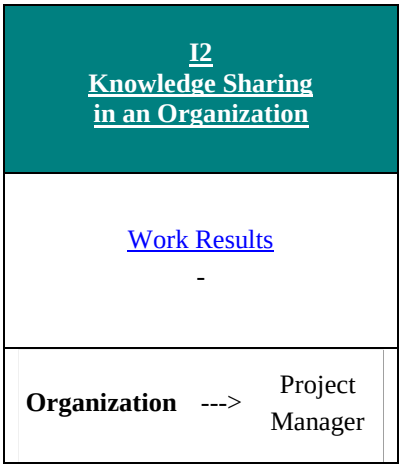
Project Alpha had many difficulties with project management because of inexperience. This pattern could have been useful in bringing some of the basic elements more systematically into place. Project Alpha had all these activities, but not in a systematic enough manner.

## References

Chrissis, M.B., Konrad, M. and Shrum, S. (2003). CMMI - Guidelines for Process Integration and Product Improvement. SEI series in software engineering, Addison-Wesley.

## 15.28 KSP26 Reuse Approach

Dimensions and Knowledge Flow:




**Problem**

No knowledge about what work results might already exist and how to find those as potential reusables for projects.

**Initial Context**

An organization wanting to establish systematic reuse of earlier produced work results.

**Roles**

*Organization* represented by a person in a role responsible for developing software engineering, products etc. The right role varies much between organizations.

*A Project Manager* etc. defining the required work results.

**Forces**

In most of the customer projects the time-to-market is very critical and if it would be possible to use, for example, earlier-produced and tested software modules it would speed up the process and to bring more reliability to the final product.

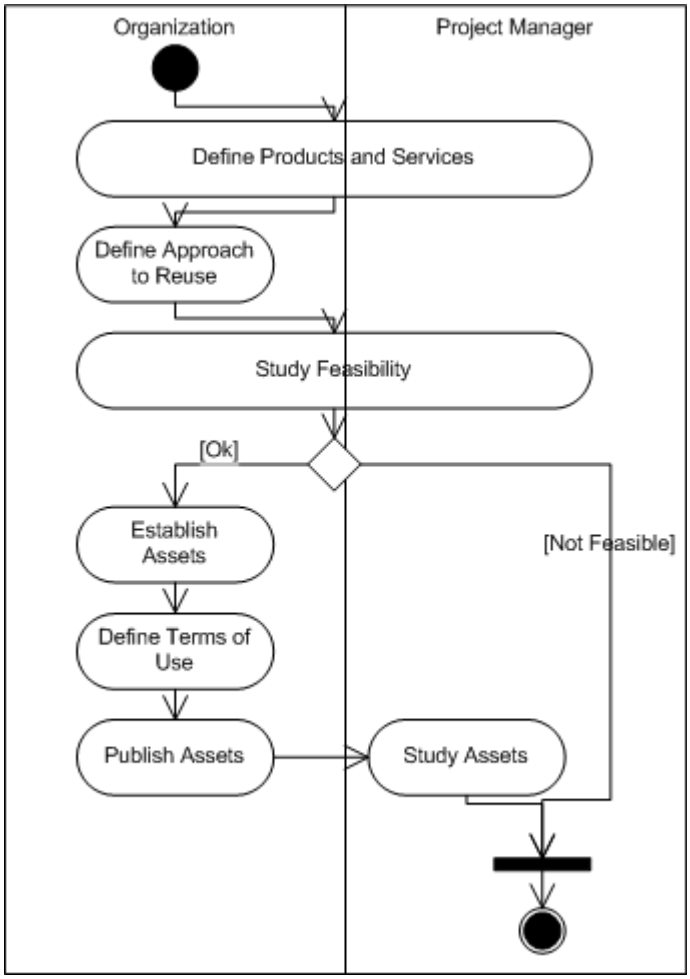
Jacobson et al. (1997, p. 6) say that through reusing software "developers can save problem-solving effort all along the development chain. They can minimize redundant work. They can enhance the reliability of their work because each reused component system has already been reviewed and inspected in the course of its original development."

Jacobson et al. (1997, pp. 8-6) define reasons why a systematic approach is required for reusing. Here are some of their topics:

- To reuse, a software engineer etc. must know what exists and could be reused.
- Traditional software development process alone is not adequate to support systematic reuse.
- Most of the software engineering organizations focus on one project at a time when reuse requires a broader focus.

- Reuse takes capital and funding.

**Solution**




1. *Define the Products and Services* etc. that you want to deliver in the future and that the markets most probably are ready to buy from you.

2. *Define Approach to Reuse*. Check from your past, what similarities there are regarding to work results. Check also the planned products/services, what similarities there could be. What could be the granularity in your reuse. Is it just some code from earlier projects or are there reasonable possibilities for large scale reuse, for example, to product-line approach. Define what could be reused and how.

3. *Study Feasibility.* Study the feasibility of your reuse approach and planned reusable assets. Do for example, cost-benefit analyses, market surveys. Make decision about the approach. The decision can also be, that the projects are so different from each other and systematic reusing will not pay off.

4. *Establish Assets*. Start the production of reusable assets according to your decisions. Define also owner(s) for the asset and the maintenance responsibilities.

5. *Define Terms of Use.* Define terms of use for reusable assets. Notice also possible IPR restrictions and warnings.

6. *Publish Assets* Publish the existing and planned assets, terms of use and contact persons. Pay attention to easy finding and use.

7. *Study Assets*. Project Managers etc. need to get familiarized with the assets at

such a level that they know what those are and when those could be applicable.

Note that this is an overly simplified model and is meant just to start reusing and thinking about the reuse approach.

**Resulting Context** Clearly defined approach to reuse. First reusable assets initiated and communicated to the organization. Organization initiated to maintain and to support the use of the assets in projects. See Quickly Made (KSP13) for how to utilize the assets in projects.

**Instances** Utilize once to initiate this kind of work and after that for each new asset. Maintenance should then be continuous during the lifetime of the asset.

One potential pitfall is reuse because of reuse and not because of business or effectivity reasons. This could lead to a situation where efforts are taken to establish reusable assets that will not have an adequate return on investment.

**Process Connection** Software engineering

## Justification

**Basic Idea** To introduce reusable work results (artifacts etc.) to be available for projects.

**Positive Instance** E.g. the process for systematic software reuse by Jacobson et al. (1997, pp. 15-17).

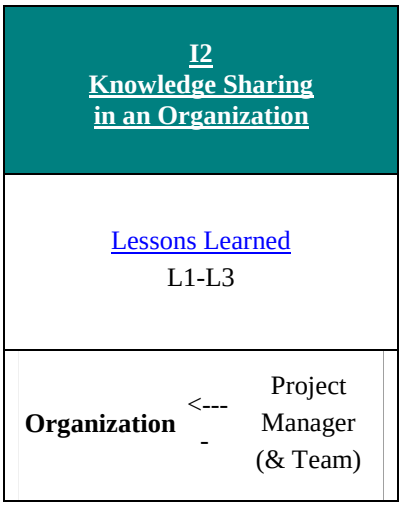
**Negative Instance** Not valid here. A project can be made also without utilizing any earlier work results.

## References

Jacobson, I., Griss, M. and Jonsson, P. (1997). Software Reuse: Architecture, Process and Organization for Business Success. ACM Press, Addison-Wesley.

## 15.29 KSP27 Contributed Experience Base

Dimensions and Knowledge Flow:




**Problem**

Lessons learned are collected in projects but not shared at the organizational level. The same problems are occurring again and again in different projects.

**Initial Context**

An organization has Established an Experience Base (KSP16). Lessons are discovered (KSP15) in project teams.

**Roles**

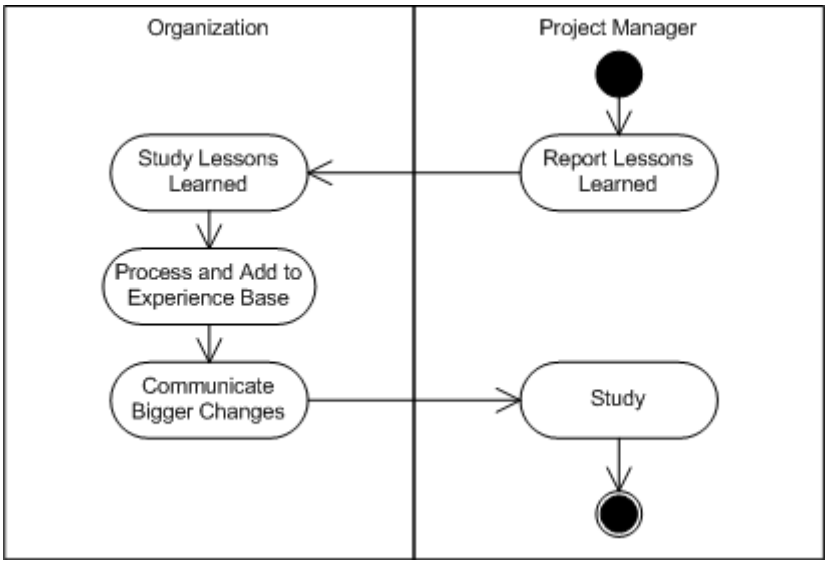
An *Organization* represented by persons responsible for the maintenance of the experience base. If the experience base is automated enough it could also represent here the organization role. *Project Manager* (including other project team members).

**Forces**

In organizations gained knowledge is "wasted" if not shared. The existing knowledge should not be difficult to find and reuse for e.g. project managers. (Komi-Sirviö et al. 2002.)

Rus and Lindvall (2002) remind of the importance of knowledge sharing in utilizing the individual knowledge at the organization. They say that: ”Large organizations cannot rely on informal sharing of employees’ personal knowledge. Individual knowledge must be shared and leveraged at project and organization levels.” (Rus and Lindvall 2002, p. 27). One important part of the organization level knowledge sharing is the lessons learned type of knowledge.

**Solution**




1. *Report Lessons Learned*. Report the lessons learned collected in a project team (see Discovered Lessons, KSP15). Instead of lessons learned, the input can also be some other feedback to be noticed in the experience base or other assets. This same procedure can be used also for the other input.

2. *Study Lessons Learned*. Do the lessons include new material for the experience base or for some other assets or forums? Do the lessons learned create a need to update the assets? etc.

3. *Process and Add to Experience Base.* Translate the material into a form that others can use (Dixon, 2000, p. 21). Update the Experience Base (or other assets).

4. *Communicate Bigger Changes*. If the previous steps have resulted in bigger changes in the experience base or other assets, inform project managers and project teams. Different ways of communication could be used for different purposes. For example, a news group to inform about the availability of some knowledge or direct communication supported with emails to everyone if very big changes happen.

5. *Study.* Follow the information related to the experience base and other possible supporting assets some reasonable way.

**Resulting Context** An organization having implemented systematic experience collecting and storing.

**Instances** Follow this pattern always when receiving lessons learned of relevant feedback related to the experience base or other such assets.

One potential pitfall is that the lessons learned could be collected and even stored into the system, but in such a way that the system can not be used for retrieving knowledge in a reasonable way.

**Process Connection** Supports process improvement and organizational development. This can affect any of the processes.

## Justification

**Basic Idea** To collect, store and share lessons learned systematically.

| | |
|---|---|
| **Positive Instance** | Leveraging common knowledge in an organization, Dixon (2000, pp. 19-21). |
| **Negative Instance** | Projects can be run without systematical lessons learned collecting and sharing, especially if an organization is very small and the lessons learned are shared as part of continuous communication. In a bigger organization, however, not having systematical experience / lessons learned sharing the same problems may occur again and again. In such a case, achieved best practices are used only by persons knowing those because of their own learning or because of someone in their unofficial network (see My Network, KSP21) is sharing voluntarily this knowledge. |

## References

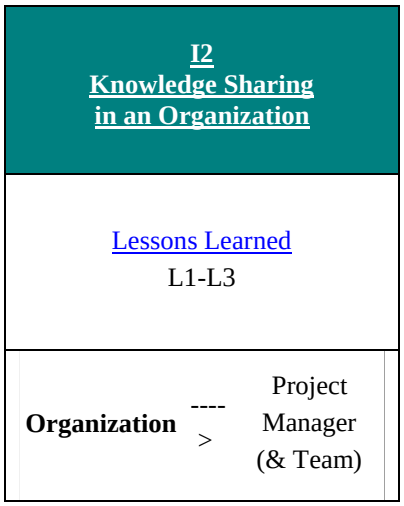
Dixon, N.M. (2000). Common Knowledge: How Companies Thrive by Sharing What They Know. Harward Business School Press, Boston, Massachusetts.

Komi-Sirviö, S., Mäntyniemi, A. and Seppänen, V. (2002). Toward a Practical Solution for Capturing Knowledge for Software Projects. IEEE Software, vol. 19, no. 3.

Rus, I. and Lindvall, M. (2002). Knowledge Management in Software Engineering. IEEE Software, 19(3), 26-38.

## 15.30 KSP28 Utilized Experience Base

Dimensions and Knowledge Flow:




**Problem**

Projects are not utilizing the lessons learned that are collected in the organization. The same problems are occurring again and again in different projects.

**Initial Context**

An organization has Established an Experience Base (KSP16). Material collection to the Experience Base has been started, see Contributed Experience Base (KSP27).

**Roles**

*Project Manager* (including other project team members) and the *Organization* represented by persons responsible for the maintenance of the experience base. If the experience base is automated enough, it could also represent the organization's role.

**Forces**

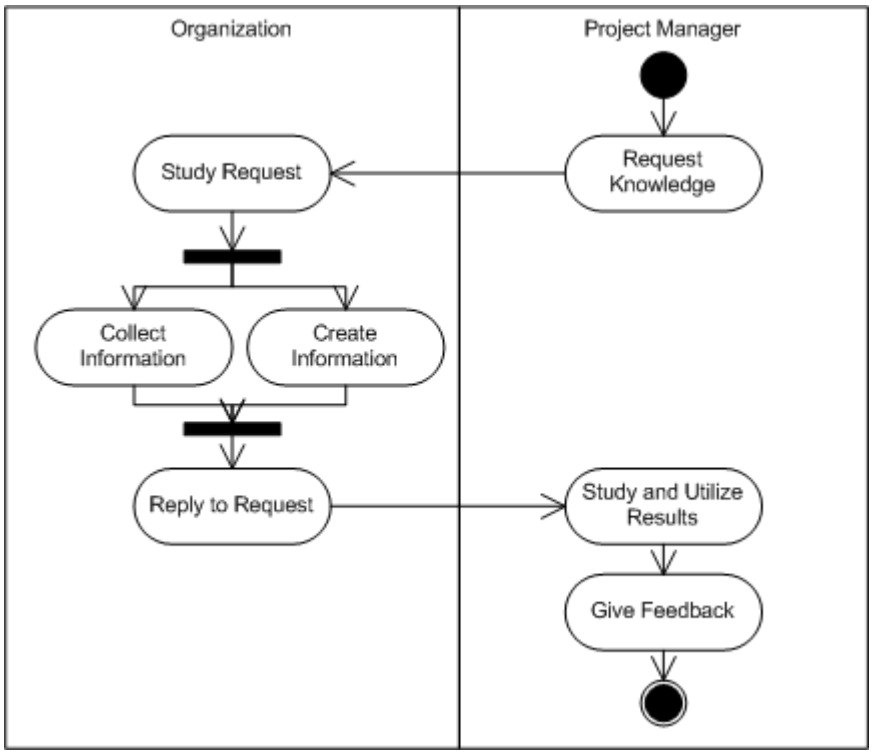
In organizations gained knowledge is "wasted" if not shared. The existing knowledge should not be difficult to find and reuse by project managers, for example. (Komi-Sirviö et al. 2002.)

Rus and Lindvall (2002) remind us of the importance of knowledge sharing by utilizing the individual knowledge at the organizational level. They say that: "Large organizations cannot rely on informal sharing of employees' personal knowledge. Individual knowledge must be shared and leveraged at project and organization levels." (Rus and Lindvall 2002, p. 27). One important part of the organization level knowledge sharing is the lessons learned type of knowledge.

Any experience system is actually not storing knowledge except information that can be used as raw material of which people create knowledge when receiving (e.g. Nonaka, 1994).

Information adapted from the experience base may create innovations or at least stepwise improvements in the use of the receiver (Dixon, 2000, p. 21). These improvements should be utilized also in the experience base.

**Solution**




1. *Request Knowledge* and *Study Request*. A project asks for knowledge. In practice, this can happen also, so that a project team member is implementing this using the user interface of the experience base. If it is something that can not be solved automatically, a human person is required to study the request and decide the required actions.

2. *Collect Information* and/or *Create Information.* If the information required for the requested knowledge is in the experience base or other assets it can be directly retrieved (automatically by the system). If not, then the information needs to be created (and updated into the experience base) or a decision is made that it is not possible to serve this knowledge request.

3. *Reply to Request.* Reply to the requester with the requested information, information close to it or reply that serving in this case is not possible.

4. *Study and Utilize Results.* Receiving person/team adapts knowledge for use in particular context (Dixon, 2000, p. 21).

5. *Give Feedback.* The person/team adapting the knowledge very often improves the solution or notices problems in the information delivered from the experience base. This knowledge should be fed back to the experience base (see Contributed Experience Base, KSP27).

**Resulting Context** An organization collecting, maintaining and sharing experience knowledge.

**Instances** Follow this pattern always when receiving a knowledge request related to the experience base or other assets.

One potential pitfall is that the experience base (or other assets) do not include the information required and the organization has insufficient resources or need to create it. If this happens very often, the motivation for using this kind of system drops.

**Process Connection** Supports process improvement and organizational development. This can affect any of

the processes.

## Justification

**Basic Idea** To collect, store and share lessons learned systematically.

**Positive Instance** Leveraging common knowledge in an organization, Dixon (2000, pp. 19-21).

**Negative Instance** Projects can be run without systematical lessons learned collecting and sharing, especially if an organization is very small and the lessons learned are shared as part of continuous communication. In a bigger organization, however, not having systematical experience / lessons learned sharing the same problems may occur again and again. In such a case, achieved best practices are used only by persons knowing those because of their own learning or because of someone in their unofficial network (see My Network, KSP21) is sharing voluntarily this knowledge.

## References

Dixon, N.M. (2000). Common Knowledge: How Companies Thrive by Sharing What They Know. Harvard Business School Press, Boston, Massachusetts.

Komi-Sirviö, S., Mäntyniemi, A. and Seppänen, V. (2002). Toward a Practical Solution for Capturing Knowledge for Software Projects. IEEE Software, vol. 19, no. 3.

Nonaka, I. (1994), A Dynamic Theory of Organizational Knowledge Creation. Organization Science, 5 (1), 14-37.

Rus, I. and Lindvall, M. (2002). Knowledge Management in Software Engineering. IEEE Software, 19(3), 26-38.

# 16 KNOWLEDGE SHARING PATTERN LANGUAGE APPLICABILITY SCENARIOS

***Scenario 1*** (Process): A new version management tool is needed in a project. The training will be arranged ten days prior to the use.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: Supporting
***Explanation***: Knowledge Sharing Pattern Language supports the identification of competences required for a project (KSP19, assuming that the tools have been remembered as one competence area) and the planning of required trainings for a project (KSP18). KSP18 is originally defined to be used only in the beginning of a project, but here it is assumed to be utilized also during project realization. Introduction of a new tool may change the software development processes in use (KSP22).
***Involved Patterns***:

- KSP19 Assured Resources
- KSP18 Improved Competences
- KSP22 Work Guidance

***Scenario 2*** (Process): Requirements from the customer need to be received at the right detail level early enough for design and implementation.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: Some Support
***Explanation***: Requirements gathering is initiated as part of defining the required project results (KSP05). This ensures that it is initiated at the earliest phase possible, when starting planning begins for a new project. KSP06 supports the analysis of the received requirements and the definition of the requirements for the project. It is a reminder of the required detail level. Knowledge sharing patterns do not fully support the validation of the user perspective (e.g. "right detail level"). Some support is received from the iterative nature of KSP05.
***Involved Patterns***:

- KSP05 Shared Understanding
- KSP06 Discovered Bones

***Scenario 3*** (Process): Very often the customer can tell how something needs to be implemented, but not the actual requirement that has caused this need. The supplier needs to find the real hidden requirements.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: No Support
***Explanation***: Requirements gathering guidance (KSP06) does not guide the supplier to question the requirements stated by the customer.
***Involved Patterns***:

- KSP06 Discovered Bones

***Scenario 4*** (Process): Ensuring that the definition of requirements will be closed when there is no added value with more details.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: Some Support
***Explanation***: Requirements gathering (KSP06) does not directly support the identification that the requirements are good enough. KSP05 brings iterative cycles to the definition of requirements including meetings with the customer. This gives some support but not clearly enough to help to understand when the definition of requirements should be closed.
***Involved Patterns***:

- KSP06 Discovered Bones
- KSP05 Shared Understanding

***Scenario 5*** (Process): Ensuring that the customer and the supplier's personnel who are implementing the project have understood the requirements for the project the same way and avoiding the rework otherwise resulting from the misunderstandings.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: Supporting
***Explanation***: KSP08 encourages using the same engineers to define the requirements together with the customer and to implement the project jointly. In addition, KSP06 reminds about the communication (& documenting) required to share the understanding of requirements in the project team and with the customer KSP05. The identified and documented requirements are reviewed and agreed upon with the customer. Of course, there is always the possibility of a misunderstanding at a more detailed level than what has been documented, but the basic practice is in place. The quality of documentation can not be ensured with these patterns.
***Involved Patterns***:

- KSP08 Created Skeleton
- KSP06 Discovered Bones
- KSP05 Shared Understanding

***Scenario 6*** (Process): Avoiding the problem: Losses of information because of language problems (e.g. different language, or different terminology). The supplier does not fully understand what the customer says or has difficulties understanding a document in the same way.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: No Support
***Explanation***: This perspective has not been covered in the Knowledge Sharing Pattern Language. It does not include patterns suggesting e.g. formalizing the documents.
***Involved Patterns***:

- None

***Scenario 7*** (Meta): The patterns need to have the right size/modularity to be easy to apply to the processes.
***Main Quality Factor***: Interoperability (Functionality)
***Result***: Some Support
***Explanation***: The patterns have been built to support defined problem areas. Through that those should have the right modularity to support solving the problem areas. Without more thorough study, however, the result can not be more than Some Support.
***Involved Patterns***:

- All patterns except I4 patterns.

***Scenario 8*** (Process): Avoiding the problem: A database synchronization problem is found at the bottom level of the organization affecting the whole organization. Knowledge about it would be very important to be shared quickly with the top management, but it is not shared properly or it is moderated on the way to the top management not initiating the decisions needed.
***Main Quality Factor***: Accuracy (Functionality)
***Result***: No Support
***Explanation***: Knowledge Sharing Pattern Language does not directly introduce any defect/risk/issue escalation patterns. KSP09 initiates risk management and KSP25 continues with the risk management but these do not refer directly to any escalation type of communication.
***Involved Patterns***:

- KSP09 Schedule Baseline
- KSP25 Followed Progress

***Scenario 9*** (Meta): The utilization of a pattern is visible from the process.
***Main Quality Factor***: Interoperability (Functionality)
***Result***: Supporting
***Explanation***: The use of a pattern is visible in a procession of steps that follow the pattern solution steps, assuming that the process is described in a way that the structure is visible.
***Involved Patterns***:

- All patterns except I4 patterns.

***Scenario 10*** (Meta): The utilization of a pattern is visible in the real life (not just in the process).
***Main Quality Factor***: Interoperability (Functionality)
***Result***: Supporting
***Explanation***: The use of a pattern is visible in a procession of steps that follow the pattern solution steps. In addition the Knowledge Sharing Pattern Language includes a pattern to support definition and use of processes (KSP22).
***Involved Patterns***:

- KSP22 Work Guidance and all other patterns.

***Scenario 11*** (Meta): Having a real life problem and having an easy way to find a solution to it from the Knowledge Sharing Pattern Language.
***Main Quality Factor***: Understandability (Usability)

***Result***: Supporting
***Explanation***: The use of the knowledge sharing interface and/or the target knowledge type to identify a potentially smaller set of patterns and then browsing the pattern problems to find the right pattern (assuming that there is such a pattern).
***Involved Patterns***:

- All patterns

***Scenario 12*** (Process): Applying the knowledge-sharing patterns results in an easy to understand and apply to process.
***Main Quality Factor***: Understandability (Usability)
***Result***: Some Support
***Explanation***: Implementation of the knowledge sharing patterns in processes will bring some clear structures to the processes, but do not automatically make the processes understandable and easy to use.
***Involved Patterns***:

- All patterns except I4 patterns

***Scenario 13*** (Meta): An existing RUP-based software development process is made to implement the knowledge-sharing pattern language in one week.
***Main Quality Factor***: Time Behaviour (Efficiency)
***Result***: Supporting
***Explanation***: The target knowledge type streams of patterns are examined one by one and the patterns there are compared to the current process (assuming that the process has been described). The process is improved where the patterns are not yet adequately implemented. The improvement can be implemented by changing the process or linking from the process to a knowledge sharing pattern.
A new knowledge sharing pattern could be defined based on the use of the KSF to identify knowledge-sharing challenges in processes.
***Involved Patterns***:

- All patterns except I4 patterns

***Scenario 14*** (Meta): Application of one single pattern in a very concrete situation solves the problem immediately.
***Main Quality Factor***: Time Behaviour (Efficiency)
***Result***: Some Support
***Explanation***: Implement the steps of the pattern solution (assuming that a required pattern is available). Initial context in the pattern supports the selection.
***Involved Patterns***:

- All patterns

***Scenario 15*** (Process): A request for a proposal (RFP) has been received from Tampere City regarding a surveillance system for parking houses. An answer to this RFP can be given after two meetings with them.

***Main Quality Factor***: Time Behaviour (Efficiency)
***Result***: Supporting
***Explanation***: Implement shared understanding of aimed results (KSP05) including definition of requirements (KSP06). The first meeting is used for understanding the customer's requirements and initiating the solution, including drafting. Earlier defined reference requirements are utilized (KSP07, KSP23, similar earlier implementation in Helsinki). The second meeting expects to have a shared understanding based on the proposal draft and improving the draft if needed.
***Involved Patterns***:

- KSP05 Shared Understanding
- KSP06 Discovered Bones
- KSP07 Reference Requirements
- KSP23 Not Wasted

***Scenario 16*** (Process): A review of a document needs to be implemented in two hours.
***Main Quality Factor***: Time Behaviour (Efficiency)
***Result***: No Support
***Explanation***: No patterns to support this scenario.
***Involved Patterns***:

- None

***Scenario 17*** (Process): A new change request is processed in a reasonable time (e.g. in four working hours) resulting in the knowledge about what needs to be done.
***Main Quality Factor***: Time Behaviour (Efficiency)
***Result***: Supporting
***Explanation***: Received change request is compared to the requirements and the requirements traceability (KSP08) knowledge is utilized to define required changes (KSP24).
***Involved Patterns***:

- KSP08 Created Skeleton
- KSP24 Flexible Skeleton

***Scenario 18*** (Process): A correction of a severe defect announced by the customer takes five days from the announcement.
***Main Quality Factor***: Time Behaviour (Efficiency)
***Result***: Some Support
***Explanation***: Some support by utilizing the requirement’s traceability (KSP08, KSP24), but no direct support for defect management. Assuming the defect can be fixed in that time.
***Involved Patterns***:

- KSP08 Created Skeleton
- KSP24 Flexible Skeleton

***Scenario 19*** (Process): The process supports reuse of architecture from an earlier project.
***Main Quality Factor***: Resource Utilization (Efficiency)

***Result***: Supporting
***Explanation***: An organization needs to have decided and implemented their approach to reuse (KSP26). Reuse possibilities are checked (KSP12) when starting the definition of work results. Existing earlier architecture is used (KSP13) in the new project (assuming that there are earlier similar projects).
***Involved Patterns***:

- KSP26 Reuse Approach
- KSP12 Managed Versions
- KSP13 Quickly Made

***Scenario 20*** (Process): The chief architect of a project leaves the company. A new, properly-educated person can effectively take his/her place in a month.
***Main Quality Factor***: Resource Utilization (Efficiency)
***Result***: Some Support
***Explanation***: The pattern language does not include very strong support for project memory in one project. The work status stream patterns (KSP09, KSP25 and KSP10) support the understanding of the project status. KSP12 ensures identification and configuration management of the work results. Some indirect support can also be obtained from the lessons learned patterns (KSP16, KSP15, KSP27, KSP28), work guidance (KSP 22) and other patterns ensuring that in a project the required results (KSP05), requirements (KSP06) etc. are well known and well documented.
***Involved Patterns***:

- KSP09 Schedule Baseline
- KSP25 Followed Progress
- KSP10 Known Status of Projects
- KSP12 Managed Versions
- KSP16 Established Experience Base
- KSP15 Discovered Lessons
- KSP27 Contributed Experience Base
- KSP28 Utilized Experience Base
- KSP22 Work Guidance
- KSP05 Shared Understanding
- KSP06 Discovered Bones

***Scenario 21*** (Process): The process supports finding earlier work results that can be reused in a new project.
***Main Quality Factor***: Resource Utilization (Efficiency)
***Result***: No Support
***Explanation***: The patterns (KSP26, KSP12, and KSP13) support the reuse in general, but do not really support the finding of earlier work results.
***Involved Patterns***:

- KSP26 Reuse Approach

- KSP12 Managed Versions
- KSP13 Quickly Made

***Scenario 22*** (Process): A new tool (e.g. Magic Draw and UML language) is taken into use. A proper guidance exists or training is arranged to have the organizational standard use of the tool shared.
***Main Quality Factor***: Resource Utilization (Efficiency)
***Result***: Some Support
***Explanation***: Through the work guidance (KSP22) some existing definitions can be shared and trained when noticing that a project team is missing these competences (KSP19, KSP18). Knowledge Sharing Pattern Language, however, does not directly support defining the organizational standards for tool use.
***Involved Patterns***:

- KSP22 Work Guidance
- KSP19 Assured Resources
- KSP18 Improved Competences

***Scenario 23*** (Meta): An organization is starting to use a new CSCW environment and the patterns automatically work in this environment.
***Main Quality Factor***: Adaptability (Portability)
***Result***: Supporting
***Explanation***: Knowledge sharing patterns work with or without these kinds of environments. A new CSCW tool will give new possibilities to implement single patterns. The contexts of the patterns do not include direct references to tools or tool uses.
***Involved Patterns***:

- All patterns

***Scenario 24*** (Process): A software engineering process utilizing the Knowledge Sharing Pattern Language has been defined for ten persons and works well. A project using that process grows from ten to two hundred persons and the knowledge sharing patterns still work.
***Main Quality Factor***: Adaptability (Portability)
***Result***: Supporting
***Explanation***: A growing project results more interfaces inside a project team. The project needs to be replanned utilizing KSP09. The knowledge sharing patterns applied in a project team (I1 patterns) need to be looked through again compared to the new setting. In general they should work, but they do not include any specific patterns that help directly in these kinds of extensions. New project team members will need training for the project and the environment. This would result in a need to check the competences of the team (especially the new members) compared to the need and planning the required training etc (KSP18). Work guidance (KSP22) would also help in sharing the ways of working with the new team members.
***Involved Patterns***:

- KSP09 Schedule Baseline

- KSP25 Followed Progress
- KSP04 Trust or Check
- KSP08 Created Skeleton
- KSP24 Flexible Skeleton
- KSP12 Managed Versions
- KSP22 Work Guidance
- KSP15 Discovered Lessons
- KSP18 Improved Competences

***Scenario 25*** (Process): Earlier the system specification has been done on three levels: architectural, sub-system and the module level. Different teams have been working with different sub-systems. Now the amount of improvement work is decreasing and the sub-system teams are replaced with one team for the whole architecture.
***Main Quality Factor***: Adaptability (Portability)
***Result***: Supporting
***Explanation***: The knowledge sharing inside the project team (I1 patterns) needs to be looked at again compared to the new setting including replanning the project utilizing KSP09. In addition, a certain amount of knowledge sharing is required to ensure that the people with extended responsibilities will have the required knowledge. Knowledge Sharing Pattern Language does not have patterns specifically for this knowledge transfer. The competence stream patterns could be applied the following way: rethinking the competence areas and resourcing (KSP19), checking the competences of the team (new reduced team) available compared to the need and planning the required training (KSP18). Having and assigning experts (KSP02, KSP03) from the other organization to help when expert help would be needed.
***Involved Patterns***:

- KSP09 Schedule Baseline
- KSP25 Followed Progress
- KSP04 Trust or Check
- KSP08 Created Skeleton
- KSP24 Flexible Skeleton
- KSP12 Managed Versions
- KSP22 Work Guidance
- KSP15 Discovered Lessons
- KSP18 Improved Competences
- KSP19 Assured Resources
- KSP03 Assigned Experts
- KSP02 Named Experts
- KSP22 Work Guidance

***Scenario 26*** (Meta): The customer requires that their processes are followed in the project.
***Main Quality Factor***: Adaptability (Portability)
***Result***: Some Support

***Explanation***: The knowledge sharing patterns can still be used, but they need to be examined and compared to the customer's processes. The differences need to be solved. Very often it means some practices will be implemented in addition to the customer processes.
***Involved Patterns***:

- All patterns except I4 patterns

***Scenario 27*** (Process): The system includes several components and the supplier decides to use a sub-supplier to implement one of the components.
***Main Quality Factor***: Adaptability (Portability)
***Result***: Supporting
***Explanation***: The knowledge sharing patterns for customer-supplier interface (I3 patterns) will be utilized so that the supplier acts in the customer role and the sub-supplier acts in the supplier role.
***Involved Patterns***:

- KSP11 Informed Customer
- KSP05 Shared Understanding
- KSP06 Discovered Bones
- KSP14 Release
- KSP22 Work Guidance
- KSP17 Satisfied Customer
- KSP19 Assured Resources

***Scenario 28*** (Process): The engineer gathering the requirements needs to know the key people in the customer organization to interview in order to gather the right requirements.
***Main Quality Factor***: Accuracy (Functionality) - defined after defining the scenario.
***Result***: Some Support
***Explanation***: Requirements gathering (KSP06) starts with the identification of the stakeholders. It, however, does not remind or specially support this situation.
***Involved Patterns***:

- KSP06 Discovered Bones

***Scenario 29*** (Process): Mr. X has programmed one very critical module for three months and has a car accident. A new person continues from this successfully.
***Main Quality Factor***: Resource Utilization (Efficiency) - defined after defining the scenario.
***Result***: Some Support
***Explanation***: The pattern language does not include very strong support for project memory in one project. Main support is received from the existing intermediate and final work results, e.g. code and documents (KSP12) and from the existing work guidance (KSP22). Some indirect support can be obtained from the lessons learned patterns (KSP16, KSP15, KSP27, KSP28) and other patterns ensuring that in a project the required results (KSP05), requirements (KSP06) etc. are well known and well documented.
***Involved Patterns***:

- KSP12 Managed Versions
- KSP22 Work Guidance
- KSP16 Established Experience Base
- KSP15 Discovered Lessons
- KSP27 Contributed Experience Base
- KSP28 Utilized Experience Base
- KSP05 Shared Understanding
- KSP06 Discovered Bones

***Scenario 30*** (Process): Avoiding the problem: The process does not suit the purposes and the people fake following the process. For example the information is stored only to emails and not where it should be.
***Main Quality Factor***: Suitability (Functionality) - defined after defining the scenario.
***Result***: Some Support
***Explanation***: The pattern language does not fully support avoiding these kinds of problems, but it has a follow-up mechanism of process use (KSP22) which could result when this kind of situation is identified and actions are initiated to correct the situation.
***Involved Patterns***:

- KSP22 Work Guidance

***Scenario 31*** (Process): Avoiding the problem: A change to the interface of a sub-system has been made, but the information about the change has not reached everyone in the project team. This causes problems in integration and requires rework.
***Main Quality Factor***: Resource Utilization (Efficiency) - defined after defining the scenario.
***Result***: Supporting
***Explanation***: As part of the processing of a change request (KSP24), the need for communication is introduced. If the change requires repeating the planning cycle (KSP09), the project team should be involved and the common planning further supports sharing the knowledge.
***Involved Patterns***:

- KSP24 Flexible Skeleton
- KSP09 Schedule Baseline