Communicating, Measuring and Preserving Knowledge in Software Development

Conny Johansson

Blekinge Institute of Technology
Department of Software Engineering and Computer Science

ISBN 91-631-0173-4

Printed in Sweden
Kaserntryckeriet AB
Karlskrona 2000

It is not the knowing that is difficult, but the doing

Chinese Proverb

This licentiate's dissertation is submitted to the Research Board at Blekinge Institute of Technology, in partial fulfillment of the requirements for the degree of Licentiate of Engineering.

Contact Information:

Conny Johansson
Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Soft Center
SE-372 25 RONNEBY
SWEDEN

Tel.: +46 457 38 58 24
Fax.: +46 457 271 25
email: Conny.Johansson@ipd.hk-r.se
URL: http://www.ipd.hk-r.se/cjh

Abstract

Software Engineering is a rapidly changing area, especially in terms of its technological foundation. The computer and information technology changes both the kind of systems to be built and the methods and tools available with which to build them. To be able to stay competitive there is no doubt that managing knowledge is very important to the corporate learning process. But even when companies are superior to their competitors technologically, they often find it hard to handle knowledge within the company.

Knowledge is to know, to be aware of something. Knowledge that have been gained by action, by exercise of a profession, is the most valuable knowledge. Explicit knowledge can be expressed by words and numbers, while tacit knowledge is not easily expressed and thus hard to formalize and write down.

This licentiate's dissertation presents the results from efforts in communicating, measuring and preserving knowledge. Approaches for communicating knowledge to individuals with no or little knowledge within the software development domain are presented. Furthermore, experiences regarding knowledge management at team level (small group) are presented. Our study show that knowledge is hard to measure, and thus difficult to preserve in text or number format. Instead, we propose that you should build social networks and rely more on oral communication.

Acknowledgments

I would like thank my advisor *Professor Jan Bosch*, for offering me this opportunity. Thanks to all my co-authors, *Lennart Ohlsson*, Utilia Consult, *Martin Höst*, Lund Institute of Technology, *Yvonne Dittrich*, University of Karlskrona/Ronneby (nowadays renamed to Blekinge Institute of Technology), *Antti Juustila*, University of Oulu, *Patrik Hall*, nowadays Cell Network AB, and *Michael Coquard*, Ericsson Software Technology AB. I would also like to thank all my colleagues at Ericsson Software Technology AB, especially *Mikael Svensson*, *Benny Rasmussen* and *Peo Sundelius* for all the support when concluding this licentiate's dissertation.

I would also like to express my gratitude to all the students at the Software Engineering program, and the People, Computer and Work program at the Blekinge Institute of Technology, who have participated with great inspire in all the projects which has been subject for this research.

Finally, thanks to my family for having patience; *Elisabeth* my companion through life, and our children *Annie* and *Albin*.

Contents

Introduction

The study of knowledge is as old as human history itself. Argyris [1] claims that the natural quality of needing continuous learning is the core of each human's development. Managing knowledge is important to our daily lives, as well as for the company's learning process. Knowledge could be transferred in several ways. From time immemorial knowledge has been transferred by oral tradition. When the art of printing was invented, knowledge was written down with the intention to be preserved for future use. With modern information technology it seems like we have good opportunities for communicating and preserving knowledge efficiently. The main problem appears to be to express the knowledge in words and numbers, which can be interpreted unambiguously by those who receive it.

Michael Polanyi turned to philosophy as an afterthought to the career as a scientist. As early as 1966 (or most probably even earlier), he discerned the tacit dimension [2]. Polanyi expresses much of the difficulties we stumbled across in our research. The core is that *we can know more than we can tell*. Even though we think we have the solution to a problem, we have hard time expressing it in words. Further, it is difficult to write down exactly what we know, and assume that it will be interpreted correctly. Polanyi takes as an example a person's face. We can recognize it among a thousand. Yet we cannot put this knowledge into words and write it down. The problem of communicating, measuring and preserving knowledge still remains, even though we are nowadays living in a world that is flooded with modern information technology.

1. Definitions and Background

This section presents a background to the research field. The most important terms are defined, and general problems are described.

1.1 Knowledge and Experience

A keen interest in the subject of knowledge has been developing during recent years [3]. Drucker [4] argues that in the new economy, knowledge is not just another resource alongside the traditional factors, but the only meaningful resource. Taking a look at a common glossary the word *knowledge* is defined as: to know, to be aware of something. Knowledge is described as to know something within a specific discipline, especially knowledge that has been acquired by studies or exercise of a profession. Some authors associate knowledge with actions [1] [5]. Sveiby [5] defines knowledge as *a capacity to act*. Davenport and Prusak [6] make this definition:

> *Knowledge is a fluid mix of framed experience, values, contextual information, and expert insight that provides the framework for evaluating and incorporating new experiences and information. It originates and is applied in the mind of knowers. In organizations, it often becomes embedded not only in documents or repositories but also in organizational routines, processes, practices, and norms.*

We have adopted these definitions when studying the kind of knowledge which is the focus in this licentiate's dissertation—experience. Experience is not only to learn something by observing or by studying. We consider experience as *knowledge that has been perceived by acting* [7]. Experiences are, according to us, the most valuable knowledge.

During the work with this dissertation we have found it important to distinguish between explicit and tacit knowledge. Explicit knowledge can be expressed by words and numbers, while tacit knowledge is not easily expressed and is thus hard to formalize [3]. Explicit knowledge can be articulated in formal language, mathematical expressions, specifications, manuals and so forth [3]. If we can identify explicit knowledge we can communicate and transfer it across individuals formally and easily. This has been the dominant mode of knowledge within the young discipline of software engineering.

Tacit knowledge, on the other hand, is hard to articulate with formal language. It is personal knowledge embedded in individual experience and involves intangible factors such as personal belief, perspective, and the value of the system [3]. Tacit knowledge incorporates so much accrued and embedded learning that it may be impossible to separate from how individuals are acting [6]. As Polanyi remarks in order to understand tacit knowledge—try to explain how you swim or ride a bicycle [2].

It is troublesome to explain the knowledge, which is hiding within measurements that at first glance seem to be explicit. Try to explain and interpret a value representing good design or planning precision. Planning precision, for example, is dependent on work methods, people's experience, unplanned technical changes, and more. In order to compare between projects, or to draw conclusions that shall be the base for future decisions, you have to ensure uniformity among processes used across the projects. Additionally you have to consider and analyse all the dependency factors. We believe that there is no doubt that tacit knowledge has been overlooked as a critical component of collective behaviour in software development [7]. Several authors report that practical knowledge, to a very large extent, is tacit [2] [3] [5] [6]. Since experience is the most valuable knowledge within the software engineering domain [7], we believe that this statement is applicable for software development as well.

1.2 Learning

According to Argyris [1] learning is not simply having a new insight or a new idea. Learning occurs when taking effective action, when experiencing something, and when detecting and correcting errors. Furthermore, Argyris gives three reasons why learning is intimately connected with actions:

- It is unlikely that what we have stored in our heads can fully cover the richness and uniqueness of a concrete situation;
- We cannot assume that other individuals will react as we have thought they would when we stored our knowledge, and that the stored knowledge will be enough to act effectively in a given situation;

- Learning is not only required in order to act effectively; it is also necessary in order to codify effective action so that it can be reliably repeated.

Peter Senge [8] shows how you can build a learning organisation with the help of five disciplines. Three of these disciplines, the first three listed below, are individual learning disciplines:

- *Systems' thinking* is the ability to understand how complex phenomenons are working as a whole, and how they influence each other;
- *Personal mastery* is to develop an own vision, and to see the environment objectively;
- *Mental models* are the insight that experiences within one area become prejudice within another area;
- *Team learning* is the collective ability among a group of individuals to develop together;
- *Building shared visions* is a discipline of translating individual vision into shared vision.

Even though the five disciplines are very much directed towards leadership and management, we think that these ideas are important for learning software development. In order to obtain a working knowledge management system you need to consider all the three levels: individual, team and organisational; which are covered by Senge's disciplines above.

1.3 Communicating Knowledge

Software companies are nowadays involved in eager attempts to manage experience and knowledge. Communication and co-ordination are extremely important issues within software development [9]. To communicate means to exchange views, which involves transfer of information and knowledge. Communication can also be explained as consultation and deliberation regarding the decision of action for solving problems. Some authors, e. g. Sveiby [5] and Davenport and Prusak [6], distinguish between information and knowledge, but we have in

our research primarily studied situations where information, containing knowledge, has been transferred with the purpose to solve problems.

Numerous attempts have been made to measure the effectiveness of various methods of transferring knowledge. They show that the most common method, the lecture, is also the least effective [5]. After five days, most people remember less than a tenth of what they heard. Learning by doing, see for example [10], is more effective—people remember 60 percent to 70 percent of what they do [5].

Modern information technology is transferring information at high speed from computer to computer. But, still, people learn mainly by following each other's example, by practising and by talking to each other [5]. New knowledge is often generated when transferring knowledge. People generally prefer to discover knowledge by experimenting. Further, craftsmen are handing down their skills through master-apprentice relationships. No one can become a master without first having been an apprentice. This kind of learning occurs in all professions and is more or less universal [5]. In software development organisations, people want immediate answers to problems, and asks more experienced people for answers [7] [11]. So even though the master-apprentice relation may not exist formally, a similar 'invisible' hierarchy exist among the developers too.

1.4 Measuring and Preserving Knowledge

Measurement is the process of assigning numbers or symbols to attributes of entities in the real world, so they can be described using clearly defined rules [12]. There exist a few approaches, which have their origin from a technical point of view, that uses measurements as a base for process improvement and knowledge management. The most famous ones are the Experience Factory approach [13], a well-known knowledge strategy among software companies, and the Capability Maturity Model, CMM, [14]. These approaches emphasise the codification approach [7] as described by Hansen et al. [15], and are characterised by knowledge management centred on the computer. Measurements shall be made with purpose to assess process performance, product quality, and the resource productivity. Furthermore, measurements are useful when calculating planning constants, which shall be used for future estimation and planning.

The problem that we, and several with us [1] [2] [3] [5] [6] [16], have encountered is how to codify knowledge without losing its distinctive properties, and turning it into unambiguous knowledge. In this process, which Nonaka and Takeuchi [3] calls externalization, tacit knowledge takes the form of metaphors, models, concepts, equations and numbers. Codified knowledge can be stored in a repository for future use by individuals within the organisation. According to Nonaka and Takeuchi [3] the experiences have to be internalized into individuals tacit knowledge bases in order to become valuable assets. Consequently, the conversion of experiences from tacit to explicit, the externalization, and the conversion back to tacit, the internalization, should result in usable knowledge originated from real experiences. The main problem is, however, to judge if the conversions gives a good result.

DeMarco [17] and Fenton and Pfleeger [12] emphasise that measurement should be an engineering activity, and that we should use measurements for controlling and steering. The objectives and goals, the measurements, which are used may differ according to the kind of personnel involved, and which level (or phase) of software development they are generated. But it is the goals that should tell us how the measurement will be used once they are collected [12].

DeMarco observe three different reason why we collect measurements [16]:

- to discover facts about the world,
- to steer our actions, and
- to modify human behaviour.

DeMarcos famous words: “You can’t control what you can’t measure”, have influenced the whole software community during more than one decade. His quote implies that we measure to steer. DeMarco has, however, later changed his mind [16]. After experiencing several examples of more or less unsuccessful measurement programs, DeMarco claims that you should measure with the purpose to discover, instead of the purpose to steer. The reason for this is mainly that we often get an unwanted decision concerning which action to use to solve the problem. As its worst, measurements can do actual harm, which case studies have reported [16] [18]. DeMarco calls this dysfunction [16], which occurs

when you measure primarily for behaviour modification and steering purposes.

One of the most famous measurement programs is the Hitachi example [18], which often, at least as we have interpreted it, is referred to as a successful program [12]. But 10 years after this program was started DeMarco reports a clear example of dysfunction [16]. Instead of reporting defects found in unit tests, the developers began refraining from reporting it. This because then their modules would appear relatively defect free, and they get less pressure later. That benefits the developer, but not the organisation.

According to Fenton and Pfleeger [12] raw measurement data should be stored in a database, which is set up using a database management system. Constraints should be defined to ensure consistency of data. Several research projects, i.e. [7] [19] [20], have attempted to design a database for software engineering data. The danger in defining general models is that it may be large, cumbersome, hard to maintain, and may not yield any useful measures [7]. Moroney [21] pointed out this danger as early as 1950.

The repository, the database, is sometimes referred to as the organisational memory [22] [23] [24]. The organizational memory is the combined owned knowledge of the organisation [22].The definition of the organisational memory differ a lot [23]. We consider it to be not only the repository of stored knowledge, but also the organisational processes, structure and culture [24].

2. Research Organisation and Results

This section gives an overview of different research methods within the area of software development. The focus and the scope of this dissertation, and the major research questions are presented. Moreover, the main contributions are clarified and grouped in relation to the research questions. Finally, we present the most important references we have studied during our work.

2.1 Method

The area of software engineering has a relatively short history, and is still immature as a research area. Research often has a formal approach as a starting point, since software engineering has emerged from mathematics and later on computer science [25]. But research in software engineering comprises not only mathematics and computer science. Economical, organisational and human and work science issues are important as well.

Four categories of software engineering research methods were proposed at the Dagstuhl workshop 1993 [26]. The categories have also been discussed by Glass [27] and Zelkowitz and Wallace [28]:

- The scientific method.
    1. Observe the world
    2. Propose a model or theory of behaviour
    3. Measure and analyse
    4. Validate the hypotheses of the model or theory
    5. Go to 1—repeat if possible
- The engineering method.
    1. Observe existing solutions
    2. Propose better solutions
    3. Build or develop
    4. Measure and analyse

5. Go to 1—repeat until no further improvements are possible

- The analytical method.
  1. Propose a formal theory of set and axioms
  2. Develop a theory
  3. Derive results
  4. Compare with empirical observations, if possible
- The empirical method.
  1. Propose a model
  2. Develop statistical or other methods
  3. Apply to case studies
  4. Measure and analyse
  5. Validate the model
  6. Go to 1—repeat

  Empirical methods can further be divided into three categories [29]:

  - An *experiment* involves the assignment of study subjects to different conditions, and manipulation of one or more independent variables. Further, measurement of the effects of the manipulation is done on one or more of the independent variables. All other variables are controlled.
  - A *survey* comprises the collection of a small amount of data in standardized form from a large number of individuals, groups or organisations from known population.
  - A *case study* involves an empirical investigation of a particular case within its real life context. Case studies involve a mix of observation, interview and analysis techniques, and both qualitative and quantitative data can be collected and analysed.

Paper I is based on the author's own experiences from developing and running an undergraduate software engineering program. We con-

sider the approach taken in Paper I to represent the scientific research method. Paper II is easily, by its title, identified as an experiment within the empirical research method. Paper III to V represents the case study research within the empirical method. Paper III also uses ethnographic research methods [9], which rely heavily on observing on-going actions, and is using a qualitative approach. Paper IV and V use more quantitative data than Paper III and are attempting at taking an organisational scope on communicating, measuring and preserving knowledge.

2.2 Focus and Scope

The research presented in this dissertation has been carried out within the software engineering domain. Software companies are often finding it difficult to handle knowledge and experiences within the company [30]. We believe that, in a research project with a scope as this dissertation, it is impossible to cover all aspects that exist, and thus impossible to build a new and complete method describing how software companies could manage their knowledge. We have observed different aspects such as company culture, the company processes and how this works on the project and the organisational level. Furthermore, by focusing on communicating, measuring and preserving knowledge, we have a reasonable chance of developing approaches and guidelines for steps towards improving knowledge management.

Aspects which have not been the focus in this work include:

- *Knowledge conversion*. The conversion of tacit knowledge to explicit.
- *Knowledge justification*. The justification and formalisation of knowledge, with the aim of using generalised knowledge across the organisation.
- *Knowledge technologies*. Knowledge engineering techniques and technologies such as expert systems and artificial intelligence.

Furthermore, our initial intentions were to not include psychological and social issues. The reason for this is probably because our educational background and professional experience is mainly within the technical area. But during our work we have moved closer to the 'soft' issues in the research for solutions, ideas and impulses for improvement.

It seems obvious to us, that in order to achieve continuous learning, knowledge management has to work at individual, team (small group) as well as at organisational levels. The five papers included have the following focus regarding this:

- Paper I. The focus is at the team level, even though the individual level is also included in order to get the team level to work.
- Paper II. The focus is at the team level, even though the result most probably will be useful on organisational level.
- Paper III. The focus is at the team level, but since there are several teams located at different geographical locations, we address the organisational level too.
- Paper IV and V. The focus is at the organisational level.

Moreover, knowledge management is about communicating knowledge to people with limited knowledge within the domain. This is the focus in Paper I. Finally; it is about knowledge management within new (inexperienced) organisations, as well as mature (experienced) organisations. Paper II and IV concerns experienced organisations, while Paper III and V concerns rather inexperienced organisations. Paper III and V are using students studying at the 4th and 6th semester on undergraduate level. Here we present several different aspects, which could be used more or less as an input to improvement, for all these categories.

2.3 Research Questions

In summary, four major research questions have formed the basis for this dissertation:

1. How can knowledge communication, primarily knowledge transferring including generating new knowledge, be improved in order to speed up learning among people with limited experience within the software engineering area?
2. How can we measure the impact specific work routines have on product quality, in our case primarily fault rate and severity, and how can we improve these work routines?
3. What are the main problems with communicating knowledge within projects and at organisational level, and how can we improve the communication?
4. How can we measure knowledge in order to preserve it for future use, especially for planning and estimating purposes for software development projects?

2.4 Main Contributions

The main results and contributions are summarised below. It includes findings which are heavily related to people's experiences, and that they learn most effectively by acting—by experience. The results and contributions are grouped in relation to the research questions of Section 2.3.

1. *Proposal for, and experiences from, a learning environment where individuals learn from actions*. The proposal includes an approach, which is strongly experienced based, and where people are allowed to make mistakes. Just like little children, you increase your skills by experimenting. It is, however, of course important that the experimentation does not jeopardize the total success of the project.

2. *An evaluation of code review methods and proposals for actions for reducing and avoiding repeating faults*. The proposals are based on experiences from two case studies, which include fault analysis. The evaluation includes an analysis of what we should do to avoid the fault from being introduced in the first place. Thus, we are aiming at preventing similar faults from being repeated over and over again.

3. *A checklist for improving group communication and an approach for building social networks for communicating experiences*. The knowledge focus related to communication is about problem solving. The checklist developed provides a framework for improving the most important issues. The approach for building social networks is based on the important finding that most experiences within the software engineering field are tacit ones, not explicit. In order to solve problems effectively we suggest a focus on mediating referrals to sources holding the correct expertise—usually human resources. Oral communication is to prefer due to the difficulty in expressing the context in writing, and the difficulty in converting tacit knowledge to explicit.

4. *Experiences from attempts in measuring and preserving knowledge*. This dissertation reports two case studies, which have an ambitious approach to measuring and preserving experiences. One case study in fact included a far-reaching design of a measurement database, which should preserve all the knowledge. The result was, however, quite different from what we initially expected. Based on our experiences, we propose to narrow the measurement scope and focus on a few measurements, such as fault data, instead. Additionally, you should develop and maintain the company culture and use person-to-person meetings in order to solve problems.

2.5 Related Work

The research addressed in this paper is related to work done around the world. This section summarises the most important references we have studied during our work. The presentation is divided into four sections, referring to Section 2.3 above.

Learning Environments

There exist numerous papers that stress the importance of creating learning environments, which emphasise learning through actions, by experience. The teamwork model used in our approach is based on the ideas of Jacquot et al. [31]. There exist similar approaches around the world, for example in USA [32], England [33], Germany [34], Finland [35] and Spain [36]. We have, however, not found any approach, which put so much emphasis on learning by doing and learning from experiences as our own.

Introducing New Work Routines and Measuring Product Quality

We have based much of our work on the measurement emphasis, which is reported in the Experience Factory approach [13] and in the Capability Maturity Model, CMM [14]. Our interpretation of the Experience Factory is that it depends heavily on the possibility of extracting explicit data, measurements. This with the purpose to formalise, generalise and store useful experiences. CMM puts strong emphasis on measuring the performance of all practices presented. Additionally, CMM has an activity within the key process area "Integrated Software Management" which says that the organisation's software process database shall be used for software planning and estimating. The new CMM version, Capability Maturity Model Integration CMMI [37], puts even more emphasis on measurements.

We have, in our research, not performed any deeper study regarding related work in introducing new work routines and measuring product quality. Höst reports in his dissertation [38] a more extensive study regarding software process impact analysis. Regarding references in measuring product quality that we have based our research on core literature within the field. Fenton and Pfleeger [12] provide, just as the title

says, a rigorous and practical approach. Another author who is recommended within this field is Jones [39] [40].

Communicating Knowledge

There exist several researchers who have done similar things like us within this area. In order to get basic knowledge within the field, we recommend the classical work by Nonaka and Takeuchi [3]. Sveiby [5] and Davenport and Prusak [6] provide two books which both are good introductions to knowledge and knowledge communication. McDonald and Ackerman [11] have performed a case study regarding expertise identification and expertise selection, which is similar to the one we have done in [7]. They have put more emphasis on empirical studies in expertise selection, while we have stronger emphasis on knowledge communication. The findings regarding expertise selection are, however, much the same.

In our work we favour direct oral communication instead of written text and numbers. We have found it obvious that modern communication technology cannot mediate all aspects of face-to-face communication. Certain technical solutions are, however, of course perfectly possible. In order to be able to identify difficulties as early as possible, we have studied the awareness problem [41] [42]. Awareness could lead to informal interactions, spontaneous connections, and the development of shared cultures.

Measuring and Preserving Knowledge

Measurements are used to assess situations, track progress and evaluate effectiveness among others [12]. The most common measures are defect tracking of bugs by users after release of software, and defect tracking during testing and inspections [12] [40]. DeMarco says [16] that he can only think of one measurement that is worth collecting now and forever: defect count. There are many other measures, but they are only worth collecting *for a while* [16]. CMM stresses defect prevention on level 5: defect prevention activities shall be planned and common causes of defects are sought out and identified [14]. These motivations, together with our professional experience from more than 10 years within industry, are strong motivations for performing research within the area.

Both the Experience Factory [13] and CMM [14] emphasise that you should build a database, an experience database with planning constants that could be used for future estimation and planning. This research was performed during four years on projects within the same organisation. Literature references within this field that we recommend is first the classical work of Boehm [43]. Further, van Vliet [25], Nicholas [44] and Royce [45], provides good introductions to the field. We have not found any comparable case study to those which are presented in this dissertation. Although there exist some cost estimation models, it is hard to find any case studies which have a pragmatic approach to cost estimation.

3. Content of the Dissertation

This licentiate's dissertation comprises five papers with different foci regarding communicating, measuring and preserving knowledge within the software engineering domain. The abstract of each paper is included in this section. Additional comments about the results—the conclusion and contribution, are described shortly. These sections are provided for the reader as a quick index and summary of results.

3.1 [Paper I] - A Practice Driven Approach to Software Engineering Education

Abstract

This paper describes a two year undergraduate education program in software engineering. This program is designed around the principle of exploratory learning, whereby the students are trained to build knowledge by themselves and actively search for solutions to the problems they experience. In addition to the essential aspects of software engineering of managing complexity of large, changing systems and the ability to work in teams, this programs also aims at preparing the students for working in a field of rapidly changing conditions and constraints. This paper describes how these high level goals have been implemented in an actual curriculum. At the core of the program is a set of project courses which are conducted as role playing games in order to simulate the conditions in an industrial environment. Students have graduated from the program for two years now, and the paper summarizes the main lessons learned as well as a follow-up survey of experiences from some of the organizations who hired the students.

Additional Comment

This paper focus on how to set up a learning environment for individuals and groups with limited knowledge within the software engineering domain. The principles have been realized for 10 years now, within the frame of an undergraduate education program. The focus is in communicating knowledge to the students. The approach is strongly experience based where people are allowed to make mistakes and learn from them.

Without having tried it, we are convinced that ideas like this is very well suited for use when training new professionals, with limited experience, within industry as well.

We found that by giving the students the freedom to experiment and learn from mistakes provides them with strong motivation to make extra efforts to their work. As early as in their first projects they need very little guidance to find answers to problems. Answers are found by asking course mates, seeking in manuals, reading literature, by the tool suppliers, or via the web. The follow-up we have made with the employers who have hired graduated students showed that the students require less internal training to become productive than students from ordinary engineering schools.

3.2 [Paper II] Evaluation of Code Review Methods through Interviews and Experimentation

Abstract

This paper presents the results of a study where the effects of introducing code reviews in an organisational unit have been evaluated. The study was performed in an ongoing commercial project, mainly through interviews with developers and an experiment where the effects of introducing code reviews were measured. Two different checklist based review methods have been evaluated. The objectives of the study are to analyse the effects of introducing code reviews in the organisational unit, and to compare the two methods. The results indicate that many of the faults that normally are found in later test phases or operation are instead found in code reviews, but no difference could be found between the two methods. The results of the study are considered positive, and the organisational unit has continued to work with code reviews.

Additional Comment

As stated above, this paper presents the results of the effects of introducing a new work routine into an organisation. The study concerns a small group working on a specific release for a product. The objective was to measure the faults found in code reviews, and to estimate the impact

they have on reducing the number of faults found in later life cycle phases. Knowledge should be acquired on team level regarding what faults that have been found, and how to avoid them from appearing again. The method we used for evaluation, including the structure of the workshop, is well suited for evaluating and measuring new methods.

We consider this study as successful, even though we could not identify any difference between the two methods evaluated. The participants in the experiment found many faults during the reviews, which they thought would be important to correct at an early stage in the development. Otherwise, the faults would remain in the system, or be found later to a higher cost.

3.3 [Paper III] Software Engineering Across Boundaries: Student Project in Distributed Collaboration

Abstract

Geographically distributed software development projects have been made possible by rapid developments primarily within the data communication area. A number of companies recognize that distributed collaboration has great potential for the near future. This report describes the empirical study of a co-operative student project located at two different geographical sites. The project was carried out at two universities, one in Sweden and one in Finland. The initial goals were to give the students the opportunity to learn about the practical aspects of co-operation between two geographically separate institutions and to study specific problems anticipated by the teachers with regard to communication, co-ordination, language, culture, requirements' handling, testing and bug fixing.

This report focuses on communication and co-ordination within the co-operative project as these were identified as the most significant problem areas. We also thought that these areas were the most interesting and the ones most likely to lead to improvements. This report not only describes our findings but also gives hints about what to think about when running similar projects both with respect to project-related issues and teaching issues.

Additional Comment

[Paper III] is, just like [Paper II], focusing on knowledge management on team level. The knowledge focus is in communicating problems, primarily regarding suggestions for solutions, and decisions which have been taken somewhere in the project. The scope is limited to project (team) level, but since the project is rather big, and sub-projects are located to different geographical locations, it could be seen as an attempt to improve communication at an organisational level too.

We consider this co-operative project to be successful and really worth the effort. We stumbled across several difficulties within the area of communicating knowledge. We developed a checklist, which provides a framework for improving the most important communication issues. This checklist is summarized below.

- Do not take your own definitions for granted. Develop a common vocabulary.
- Develop a company value. Give greater emphasis to the commitment culture [46].
- More direct contact must be taken. Provide the awareness of each other.
- Introduce the role of a virtual member of the distant group. This member should keep track of interfaces and act as a gatekeeper to disseminate relevant information for the distant group.
- Provide not only negative feedback, but also positive.
- Use a co-operative system of support and a common workspace, but remember that this does not guarantee good communication.

3.4 [Paper IV] 'Talk to Paula and Peter - They are Experienced' - The Experience Engine in a Nutshell

Abstract

People working in the Software Engineering field must solve problems every day to get their work done. For successful problem solving, they need to find others with specific experience. To help individuals with this, we created an 'Experience Broker' role in the organization. The broker must be officially recognized, and function as a facilitator for the internal human network in the organization. Our approach to the 'Experience Factory', what we call the 'Experience Engine', focuses on mediating referrals to sources holding the correct expertise—usually human sources.

We claim that the most valuable experience is tacit and stored on the individual level. The most valuable experience transfer comes while on the job when the person holding the experience verbally relates to the learner directly. Getting the right context for this experience transfer is difficult without direct contact between the person with the experience and the recipient. Experience is transferred at the right time when the learner is actually working on something that concerns it.

Values must be rooted in the organization, where individuals are willing to spend the necessary (most often short) time to spread valuable experiences to others in the organization. Our pilot project showed that a measurement and database approach is less valuable than individual face-to-face meetings, to transfer experience.

Additional Comment

In this paper we have extended the view to an organisational level. We started our work trying measuring and storing knowledge, but ended with the finding that most experiences within the software engineering field are tacit and not explicit ones. The result of the research is an approach, which heavily relies on building social networks for communicating experiences. Instead of storing all knowledge in a database on a computer, we recognized that the brain was the best primary storing media. We realized that it is very difficult to codify knowledge and recommend that you narrow the measurement program for a software

organisation. There are many difficulties associated with converting tacit knowledge to explicit, to describe the context in text, and to write down proposals for actions. Our conclusion is that there is a high probability that the knowledge would be misinterpreted, and favoured oral communication instead.

3.5 [Paper V] Surprising Results from a Measurement Study - Is Software Measurement an Exaggerated and Over-emphasised Area?

Abstract

Many software companies are involved in eager attempts to manage experience and knowledge on an organisational level. Experience that can be used for estimating and planning purposes are of great value. One obvious solution can be defining and storing a set of measurements representing these experiences, carry out statistical analyses, and developing rules and guidelines to find and use this explicit knowledge. This four-year study using student projects indicates that the great majority of these measurements, seemingly unambiguous, include several uncertainties. We found it difficult to generalise the measurements since many factors have a different influence on each. The experience database we designed contains only three planning constants. The part of the study adding significant value to process improvement efforts was the analysis of software faults found in testing. Software faults were analysed for description, consequence, and proposals for actions to avoid similar faults re-occur in the future. The potential for improvement resulting from the actions proposed is deemed high, and can probably reduce the number of major faults later on. Statistical analyses did not, however, prove useful because of all the uncertainty factors. Detailed analysis, primarily performed in seminars and face-to-face discussions, was necessary to interpret and understand the measures and statistical analyses. We are sceptical about using statistics due to all the difficulties associated with interpreting each measure, and since it is too tricky to perform a complete and fully satisfactory analysis of all existing uncertainty factors.

Additional Comment

This paper is focusing on measuring and preserving knowledge on an organisational level. The study has the purpose of building a database with planning and estimation constants, which could be used by all software development projects within an organisation. Starting off with expectations of concluding with a large amount of knowledge to store in the experience database, we were surprised finding only three reliable planning constants that could be used on organisational level. Further, we present one successful study regarding measurements—a study regarding analysis of faults. This study gave us a base for improvement, and general proposals for preventing similar faults from appearing again. Important to notice is that measurements are very hard to interpret without any detailed analysis of all factors that influence each measurement differently. Examples within the software engineering domain of such uncertainty factors are requirement changes, late hardware deliveries, and unplanned technical changes.

4. List of Papers

Below is a list of the papers included in this dissertation, along with their publication status:

[Paper I] **A Practice Driven Approach to Software Engineering Education**
L. Ohlsson, C. Johansson
IEEE Transactions on Education, Vol. 38, No 5, 1995, pp 291-295.

[Paper II] **Evaluation of Code Review Methods through Interviews and Experimentation**
M. Höst, C. Johansson
Journal of Systems and Software, Vol. 52, Issue: 2-3, June 1, 2000, pp 113-120.

[Paper III] **Software Engineering Across Boundaries: Student Project in Distributed Collaboration**
C. Johansson, Y. Dittrich, A. Juustila
IEEE Transactions on Professional Communication, Vol. 42, No 4, December, 1999, pp 286-296.

[Paper IV] **'Talk to Paula and Peter - They are Experienced' - The Experience Engine in a Nutshell**
C. Johansson, P. Hall, M. Coquard
Accepted for publication in G. Ruhe, F. Bomarius (ed.): Learning Software Organization - Methodology and Applications. Springer-Verlag. Lecture Notes in Computer Science, Volume 1756 (appears 4th quarter 2000).

[Paper V] **Surprising Results from a Measurement Study - is Software Measurement an Exaggerated and Over-Emphasised Area?**
C. Johansson
Published on CD-ROM in the 3rd European Software Measurement Conference FESMA-AEMES 2000, October 2000, Madrid, Spain

5. Related Publications

The following publications are related but not included in the dissertation:

1. An Attempt to Teach Professionalism in Engineering Education
   L. Ohlsson, C. Johansson
   Proceedings of the 3rd World Conference on Engineering Education 1992, Vol. 2 pp 319-324

2. Teaching Object-orientation: From Semi-colons to Frameworks
   C. Johansson, L. Ohlsson, P. Molin
   Proceedings of Software Engineering in Higher Education, Southampton, 1994, pp 367-374

3. Maturity, Motivation and Effective Learning in Projects - Benefits from using Industrial Clients
   C. Johansson, L. Ohlsson, P. Molin
   Proceedings of Software Engineering in Higher Education, Alicante, 1995, pp 99-106

4. Early Practise and Integration - The Key to Teaching Difficult Subjects
   C. Johansson
   Proceedings of International Symposium on Software Engineering in Universities, ISSEU ´97, Rovaniemi, 1997, pp 93-100

6. Further Work

The work, which is presented in this dissertation, includes several different aspects, which may be subject for further investigation. Currently I find knowledge management at the organisational level most interesting and challenging. Most aspects could, however, be studied both on project level (small group), and on organisational level. Some of these aspects are:

- What kind of knowledge could be considered as explicit within the software engineering domain? If we shall concentrate on physically storing explicit data, then we have to categorize and exemplify explicit knowledge.
- Must we convert tacit knowledge to explicit knowledge in order to have any use of it? Nonaka and Takeuchi [3] discuss the conversion of tacit knowledge to explicit knowledge. But do we need this conversion in order to have any use of the knowledge? Can we rely more heavily on socialization [3], that is to transfer and convert knowledge directly between individuals, instead?
- This research has mainly concerned interpretations and justification of knowledge for use on individual and limited size group level, even though the intention in the two last papers is to use the knowledge on organisational level. If experiences shall be used on organisational level we need justifications. To a wider extent apply this justification using the approaches and ideas presented in this dissertation would be very interesting.
- To calculate the cost benefit for improvement steps in knowledge management is very difficult. It would, however, be interesting to make attempts to find or develop models for doing this.
- It is quite obvious that you should not rely only on codification, which is to store measurements physically on a computer device [15]. Likewise it is obvious that you should not rely only on personalization, where knowledge is shared person-to-person [15]. A more extensive research than performed in this dissertation is necessary in order to find the balance between the two opposites.

References

[1] C. Argyris, "*Knowledge for Action*", Jossey-Bass Inc. Publishers, 1993

[2] M. Polanyi, "*The Tacit Dimension*", Routledge and Kegan Paul, 1966

[3] I. Nonaka, H. Takeuchi, "*The Knowledge-Creating Company*", Oxford University Press, 1995

[4] P. F. Drucker, "*The New Productive Challenge*", Harvard Business Review, Nov.-Dec. 1991, pp 69-79

[5] K. E. Sveiby, "*The New Organizational Wealth*", Berrett-Koehler Publisher Inc., 1997

[6] T. H. Davenport, L. Prusak, "*Working Knowledge: How Organizations Manage What They Know*", Harvard Business School Press, 1998

[7] C. Johansson, P. Hall, M. Coquard, "'*Talk to Paula and Peter - They are Experienced' - The Experience Engine in a Nutshell*", in G. Ruhe, F. Bomarius (ed.): Learning Software Organization - Methodology and Applications. Springer-Verlag. Lecture Notes in Computer Science, Volume 1756 (appears 4th quarter 2000)

[8] P. M. Senge, "*The Fifth Discipline: The Art and Practice of the Learning Organization*", Currency Doubleday, 1994

[9] C. Johansson, Y. Dittrich, A. Juustila, "*Software Engineering Across Boundaries: Student Project in Distributed Collaboration*", IEEE Transactions on Professional Communication, Vol. 42, No 4, December 1999, pp 286-296

[10] J. Dewey, "*Experience and Education*", MacMillan Publishing Company, 1997

[11] D. W. McDonald, M. S. Ackerman, "*Just Talk to Me: A Field Study of Expertise Location*", Proceedings of CSCW '98, 1998, pp 315-324

[12] N. E. Fenton, S. L. Pfleeger, "*Software Metrics, A Rigorous & Practical Approach Second Edition*", International Thomson Computer Press, 1997

[13] V. Basili, G. Caldiera, H. D. Rombach, "*Experience Factory*", Encyclopaedia of Software Engineering Vol. 2 Editor: J. Marciniak, John Wiley and Sons Inc., 1994, pp 528-532

[14] M. Paulk, C. V. Weber, B. Curtis, M. B. Chrissis, "*The Capability Maturity Model: Guidelines for Improving the Software Process*", Addison-Wesley, 1995

[15] M. Hansen, N. Nohria, T. Tierney, "*What's your Strategy for Managing Knowledge?*", Harvard Business Review March-April 1999, pp 106-116

[16] T. DeMarco, "*Why does Software Cost so Much? And other Puzzles of the Information Age*", Dorset House, 1995

[17] T. DeMarco, "*Controlling Software Projects*", Yourdon Press, 1987

[18] D. Tajima, T. Matsubara, "*The Computer Software Industry in Japan*", IEEE Computer, 14(5), 1981, pp 89-96

[19] P. Comer, P. Bradley, N. Ross, "*Software Data Collection Manual*", Issue 2, report produced by ALV/PRJ/SE/078: Software Data Library Project, 1987

[20] N. Ross, "*The Collection and Use of Data for Monitoring Software Projects"*, in B. Kitchenham, B. Littlewood (ed.) Measurement for Software Control and Assurance, Elsevier, 1989, pp 123-54

[21] M. J. Moroney, "*Facts from Figures*", Penguin books, 1950

[22] M. Markkula, "*Knowledge Management in Software Engineering Projects*", Proceedings of the 11th Conference on Software Engineering and Knowledge Engineering, June 1999, pp 20-27

[23] M. S. Ackerman, C. Halverson, "*Considering an Organization's Memory*", Proceedings of CSCW 98, 1998, pp 39-48

[24] J. P. Walsh, G. R. Ungson, "*Organizational Memory*", The Academy of Managmenet Review 16 (1), 1991, pp 57-91

[25] H. van Vliet, "*Software Engineering: Principles and Practice*", John Wiley and sons, 1993

[26] W. R. Adrion, "*Research Methodology in Software Engineering*", in Tichy, Habermann, Prechelt (ed.) Summary of the Dagstuhl Workshop on Future Directions in Software Engineering, ACM Software Engineering Notes, SIGSoft, 18 (1), 1993, pp 36-37

[27] R. L. Glass, "*The Software Research Crisis*", IEEE Software, November 1994, pp 42-47

[28] M. V. Zelkowitz, D. R. Wallace, "*Experimental Models for Validating Technology*", IEEE Computer, May 1998, pp 23-31

[29] C. Robson, "*Real World Research*", Blackwell Publishers, 1993

[30] G. Bhatt, "*Managing Knowledge through People*", Journal of Knowledge and Process Management, Vol. 5, No. 3, 1998, pp 165-171

[31] J. P. Jacquot, J. Guyard, L. Boidot, "*Modelling Teamwork in an Academic Environment*", Proceedings of the 4th SEI Conference on Software Engineering Education, pp 110-122

[32] Carnegie Mellon University, Institute for Software Research International. PHD Program in Software Engineering, http://www.isri.cs.cmu.edu/, link to Phd Program

[33] Z. Mahmoud, "*Issues in the Teaching of Software Engineering*", ISSEU '97, Rovaniemi Polytechnic, 1997, pp 125-131

[34] J. Ludewig, R. Reissing, "*Teaching What They Need Instead of Teaching What We Like - the New Software Engineering Curriculum at the University of Stuttgart*", Information and Software Technology 40, 1998, pp 239-244

[35] A. Juustila, T. Molin-Juustila, J. Kokkoniemi, "*Evolution of an Undergraduate Course on Software Engineering*", ISSEU ´97, Rovaniemi Polytechnic, 1997, pp 101-108

[36] N. Juristo, E. Tovar, "*Presence of Software Engineering in Spanish University Computer Science Curriculums*", in Software Engineering in Higher Education II, Computational Mechanics Publications, 1996, pp 161-176

[37] Carnegie Mellon, Software Engineering Institute. Capability Maturity Model Integration, CMMI. http://www.sei.cmu.edu/cmmi

[38] M. Höst, "*Early Evaluation of Software Process Change Proposals*", Doctoral Thesis, Lund University, 1999

[39] C. Jones, "*Applied Software Measurement: Assuring Productivity and Quality*", 2nd edition, McGraw Hill, 1996

[40] C. Jones, "*Software Quality: Analysis and Guidelines for Success*", International Thomson Computer Press, 1997

[41] P. Dourish, S. Bly, "*Portholes: Supporting Awareness in a Distributed Work Group*", Conference Proceedings on Human Factors in Computing Systems, 1992, pp 541-547

[42] E. Rønby Pedersen, T. Sokoler, "*Awareness Technology: Experiments with Abstract Representation*" In Proceedings of the HCI International '97, San Francisco, 1997. Elsevier Science Publishers, 1997

[43] B. W. Boehm, "*Software Engineering Economics*", Prentice Hall, 1981

[44] J. M. Nicholas, "*Managing Business and Engineering Projects: Concepts and Implementation*", Prentice Hall 1990

[45] W. Royce, "*Software Project Management: a United Framework*", Addison-Wesley, 1998

[46] W. S. Humphrey, "*Managing Technical People*", Addison-Wesley, 1997

PAPER I

A Practice Driven Approach to Software Engineering Education

Lennart Ohlsson and Conny Johansson

Published in the IEEE Transactions on Education, Vol. 38, No 5, 1995, pp 291-295

Abstract

This paper describes a two year undergraduate education program in software engineering. This program is designed around the principle of exploratory learning, whereby the students are trained to build knowledge by themselves and actively search for solutions to the problems they experience. In addition to the essential aspects of software engineering of managing complexity of large, changing systems and the ability to work in teams, this programs also aims at preparing the students for working in a field of rapidly changing conditions and constraints. This paper describes how these high level goals have been implemented in an actual curriculum. At the core of the program is a set of project courses which are conducted as role playing games in order to simulate the conditions in an industrial environment. Students have graduated from the program for two years now, and the paper summarizes the main lessons learned as well as a follow-up survey of experiences from some of the organizations who hired the students.

1. Introduction

Few areas of education pose the challenges software engineering does when it comes to selecting curriculum contents that will be of value

throughout the students professional careers. Not only is software engineering a young area, but it is also a rapidly changing area in terms of its technological foundation: computer and information technology. This technology changes both the kind of systems to be built by software engineers and the tools available with which to build them. Progress in the tool support available to software engineers is perhaps the best illustration of the changing conditions in the field. We are in the midst of a transition from the traditional editor-compiler-debugger tool set to multiuser, repository based IPSEs (Integrated Project Support Environments) with full life cycle CASE coverage.

The set of professional abilities which is considered crucial then change as well. This makes the danger of providing skills and knowledge that soon may become obsolete an important issue in the design of a software engineering program. It is not the case to the same extent in computer science where there are many problems like language translation, search efficiency and resource management, that are more long-lived. From a software engineer's perspective however, these problems tend to move away to the domain of specialists who provide tools and abstractions on a higher level.

Software engineering also has its "essential" problems [1] like managing complexity, including communication difficulties within teams, and maintaining consistency in changing systems. Providing the abilities to deal with these problems should obviously be the core of any software engineering education program. But the "accidental" problems, which are the ones that are changing so rapidly, still put constraints when it comes to the implementation of such a program. In order to teach the essential skills we must use the conceptual and computerized tools that are available today. Furthermore, the accidental problems should not be neglected, because software engineers need the ability to solve them.

It appears as if a compromise has to be made when designing a software engineering education program. We can either de-emphasize the accidental problems or we can teach the solutions available today. In this paper we describe an attempt to resolve this dilemma by designing a practice driven education program where the students are trained to learn by themselves rather than being taught canned knowledge from books and lectures.

2. Practice Driven Learning Principles

Traditional education practices seem to be built on an assumption that the mind is a container which it is the teachers' responsibility to fill with knowledge. As pointed out by Gang [2], this view can be traced back to the era of industrialization when massive public education was born. Knowledge is dissected into separate categories and students march, as on an assembly line, from one class to another where specialized operations are performed. The responsibility to integrate the various pieces of knowledge to a meaningful whole is left to the students. Both empirical studies [3] and intuition say that such integration does not come naturally.

Learner based teaching means that education is not viewed as a process where knowledge is transferred from the teacher to the student but rather that knowledge is created within the students' minds. Courses in engineering education generally consist of both lectures and hands-on laboratory exercises. The course content is traditionally defined by the lecture series with the exercises giving additional support. The underlying model is that theory should be first taught and then verified in practical exercises. We have adopted a more practice driven education where theory is regarded as something which cannot be taught but must be built by each individual. In our approach we therefore have the hands-on exercises to define the course contents and regard the lectures as supporting activities. This framework, that goes from the concrete to the abstract, capitalizes on the innate human desire to explore and learn and encourages learning that is characterized by "practice-pull" rather than "theory-push".

It should be noted that the emphasized practical work is not meant to be at the expense of abstract and theoretical reasoning. Such level of knowledge is a superior form of knowledge, but only if the student has learned it deep enough to be able to apply it on non-schoolbook problems. By obtaining abstract knowledge through guided reflections on one's own experiences this knowledge will be acquired on a higher level of understanding than can be done from a book.

Working with the premise that a person can learn only by himself has quite an effect on the role of the teacher. It requires taking a very humble attitude toward teaching. Rather than just applying an analytic knife and cutting the area of study into pieces small enough for the students to swallow, the main task of the teacher is to encourage students

to take initiatives on their own. Taking a seemingly more passive role in class does not, however, reduce the teacher's total effort. On the contrary, maintaining educational progress in this role in class requires additional effort to be spent on preparation.

A prerequisite for a more learner-centered education requires a change in the currently distributed responsibility for engineering curricula over a large number of departments. A teacher must not only know the subject he is teaching but also how this fits into the larger picture. In a situation of choice, a student is aided more in his learning process by a teacher who has an adequate knowledge of the context of his area than very specialized knowledge of his particular domain of expertise.

3. Curriculum Overview

Our program is a two year program leading to a University Certificate degree which is offered by Swedish universities as a complement to the four and a half year degree Masters programs offered at the traditional engineering schools. The students thus have no preparatory computer science or other academic training when they enter this program.

The principle of practice driven education is applied both on the level of individual courses and on the curriculum level. The large proportion of the curriculum consists of project courses (40%), and most of the other courses are designed around laboratory exercises.

To promote and utilize the potential of the exploration which learner do spontaneously, it is also crucial to have a comprehensive work environment. From the first course, the students have access to Unix workstations, and throughout the courses they are successively introduced to a number of modern tools.

Given below is a summary of all the courses in the program with an emphasis on how our pedagogical foundation has been implemented for the subject area of software engineering.

3.1 1st Semester

Computer Science

An important criteria in the choice of the primary programming language in a software engineering education is, that it is or is likely to become widely accepted in industry. On the other hand, the language should provide reasonable support for modern programming practices and be compatible with tools like database management systems, user interface management systems, etc. C++ is the language which currently best fits these criteria. By using C++ in this five week introductory programming course we have been able to introduce the object-oriented philosophy as the first way the students learn to think about programming.

The teaching of the programming language is inspired by the way humans learn natural languages, by absorption and imitation. Before the students are asked to express something that requires a new language feature, they will generally have read it several times. From the beginning the students are presented with a fairly large program where difficult parts initially are hidden by abstraction barriers. By doing carefully scoped exercises that modify this example in various ways, the students' active vocabulary gradually is extended to cover the whole language. Embedded in the example are also instances of standard algorithms and data structures which the students then encounter during the exercises. The method resembles the one used in Abelson et al. [4]. Apart from the improved motivation that comes from working with more realistic examples, this method has the benefit that the abstraction principle is learned not as yet another technique but as a natural and obvious way to manage complexity. Always programming in a context of other modules also promotes a habit of using, and later re-using, existing software.

Written and Oral Presentation

Presentation skills form the basis for producing quality documentation of software systems as well as working together in a team. The written word is the vehicle for collective thought and a means to assure that ground covered in a project is not lost. This is particularly important for a software engineer since software essentially is various forms of descrip-

tions of a system, and the basic principles for making understandable presentations are the same as those for making understandable software.

In this course the students give a number of mini-lectures and hand in some written assignments on which they receive individual feedback on disposition, style and overall clarity. The main written assignment consists of writing a beginners manual to the Unix system, which also serves as a vehicle for the students to learn the environment by themselves.

Analysis and Design Methods

Like the computer science course, this course also uses a large example. Again, the size is used to motivate the use of methods and notations for analysis and design. The students are asked to do a modification of such complexity that it is very hard to do on the code level directly. With a system that is documented in a comprehensive analysis and design notation, in our case OMT [5], the task is manageable. Through further exercises, the students progress from understanding the notation to using it on systems they design themselves.

Digital Design and Computer Organization

In this course, which covers the low-level end of the software engineering spectrum, the students work through two lab series. In the first series they use Boolean algebra and finite state machines to construct combinatorial and sequential digital circuits, ending by building a simple CPU. In the second series they work in assembly language on an HC11 microprocessor system to explore the principles of computer organization. In the final laboratory the students build a floppy disc controller, which is later used in the operating systems course.

3.2 2nd Semester

Discrete Mathematics

The purpose of this course is to give the mathematical basis of programming. Although it is the most theoretical course in the program it is still based on laboratories and practical exercises. By using a mathematical

software package, Mathematica, and conventional programming exercises, the students solve problems in set theory, graph theory, combinatorics, matrix theory and do simulations of finite state machines and Turing machines. Basic concepts like sets, tuples, mappings, trees and graphs are defined and applied as abstract data types in practical programming exercises on concrete problems. Possible trade-offs in the implementations are studied. This serves as motivation for learning complexity theory as a tool to make such trade-offs correctly.

Operating Systems and Real Time Systems

This course contains three series of laboratory exercises. The first part uses the floppy disc controller built in the course on Computer Organization and extends it to become a small file system. The second series uses Ada tasking, both to illustrate high level constructs for concurrency management and real-time programming and to build a simulation model of a virtual memory system. This model is used to study concepts like thrashing, swapping and page replacement strategies. The third part studies file and process management in Unix including X11 window system.

Individual project

This course is the first of three project courses in the program. To give some realism to these courses they are run as role playing projects, similar to the approach described in Jacquot et al. [6]. These courses also try to introduce a professional attitude. A professional is distinguished by an ability to make assertions regarding the cost and time required to deliver a product or service and to maintain a responsibility to fulfil this promise. A professional is also able to deal with those unfortunate, but inevitable, situations when he is unable to fulfil his commitment. In those situations he takes responsibility not only to minimize his own losses, but also to initiate negotiations to find a solution which, under the new circumstances, minimizes the consequences for the client. This behaviour is the natural result of an active commitment as opposed to being more passively assigned a task, as is the common situation in education environments. We have introduced the concept of 'commitment culture' for this style of working where commitments, or active and voluntary responsibility, are used as a driving force in everyday work [7].

This course runs full time during five weeks and the size of the programs delivered is approximately 5000 lines of code. The students are shown a number of existing applications and asked to do "one of those". Stating the assignments in this way naturally motivates the students to spend substantial effort on early activities like requirements analysis. The applications range from games to simple spreadsheets to process kernels for embedded systems. In the first course, the customer is only the user of the program. The required delivery only includes an executing program and user documentation, unless additional deliverables are agreed upon in negotiations.

The student's first task is to make a specification of the application which can form the basis of an agreement with the customer and to negotiate this with the customer. The negotiations not only involve specification of the product, but also the acceptance procedure in order to minimize the risk of deadline overrun, i.e. not passing the course. Things like early delivery and acceptance of a specification document in a graphical notation or of a functionally incomplete (but debugged version) of the program, may be agreed upon as part of this procedure.

The voluntary aspect of a commitment is emphasized throughout the course. The students are encouraged to re-negotiate their tasks by the customers/teachers. In this first course the students are rather complaisant negotiators, but the breaking of an agreement is however never accepted. Not even absence with illness is accepted if the customer has not been promptly informed.

In addition to teachers playing customers, there is also the role of technical advisor. No teacher plays the role of both customer and technical advisor to the same student in order to make roles as distinct as possible.

3.3 3rd Semester

Basic tools and techniques

There are several basic tools and techniques that are part of a modern software engineer's toolbox. With our condensed curriculum we cover four such areas in one course:

- Database design

- Human computer interaction
- Translation tools and techniques
- Data communication and distributed systems

The database part starts by introducing a modern, powerful development environment (INGRES) for these kinds of systems. It is introduced by presenting a complete and running application. The students learn the environment by modifying the example in various ways. This is supported by lectures on conceptual modeling and relational database design theory. The final exercise involves the modeling of a rather large organization and designing and implementing an information system for it emphasizing documentation and performance optimization.

The human-computer interaction part of the course partly uses the same examples as the database part and partly uses a stand-alone user interface management system (GUIDE). Ergonomic rules for the design of menus, forms and reports are studied, and existing style guides are compared with the user interfaces of various commercial products.

The part on translation tools and techniques starts by using the pattern-scanning language awk to do simple reformatting and translations of files. Its limitations then motivate the introduction of the more powerful tools lex and yacc. Different classes of grammars are studied and the tools are used to build a simple command tool.

The part on communication and distributed systems is done in the framework of the layered ISO/OSI model. We selected two levels for the practical exercises. First, a file transfer protocol between a Unix-workstation and a micro-controller is implemented. Second, a client-server application is built on the RPC-mechanism and its associated development tools.

Group project

In this course the students work in teams of four to five during ten weeks. The course consists of two parts: the first part is the development of a new system, and in the second part the teams are asked to make certain modifications to those systems. The parts are both run as complete projects with development task approximately twice the size of modification task. Examples of systems built in this course are:

- An on-line measurement data collection system including sensor interfaces on single board computer and transfer to a database on a workstation for presentation to the user.
- A host/target development system for the HC11, including an assembly level debugger on a workstation.

The goal of this course is to introduce informally project planning as a refinement to making commitments. The small team size in this course makes it possible to work in an intuitive, informal organization. Just working together, however, always poses problems which the students have the opportunity to solve on their own. This creates the appreciation basis for a more strict organization which is introduced in the final project course.

The project management in this course practices tracking and successive planning. Achieving quality, i.e. customer satisfaction, must be addressed throughout the project lifetime. The key is to achieve mutual commitment, to have the customer involved in order to get continuous feedback and, hopefully, a blessing of the approach taken. Minuted meetings with the customer are used to formally agree progress, and the issue of quality control is transformed to the problem of having sufficient information at these meetings in order that the customer can make competent decisions.

In this course the teacher's role of technical consultant is augmented with a function of a quality controller. His task is to assist the teams to organize their work internally, but his role is only to give advice and let the students decide for themselves how they want to work. He participates by being present at the groups' meeting once a week.

Including a modification task in the course motivates development in such a way that it facilitates future unknown changes.The problems of maintenance are made realistic and very concrete by having the groups switch systems so that everybody is confronted with code and documentation developed by someone else. The need for different kinds of system documentation is thereby further illustrated.

Project organization and human work science

How to work together in a team is not explicitly taught in the Group Project course. Instead this non-technical course is given in parallel. It highlights topics like planning, distribution of work, leadership, conflict

management, compromises and communication difficulties. The concrete experiences from the Group Project course provide motivation and reasons to actively discuss and find solutions to these issues. This course also covers the concept of commitment in more detail and its basis in that everybody must take active and voluntary responsibility to find and give information and to maintain an open and communicative atmosphere.

3.4 4th Semester

Large project

In this course, which runs for 15 weeks, the students work together in teams of 12-15. The first year this course was run meant that all student was on the same team. The project is divided in a pre-project phase and a development phase. During the pre-project phase, which constitutes about one third of the course, the problem is defined and the requirements are determined.

The development model used is based on evolutionary delivery [8], which has the following characteristics:

- Planning for multiple objectives, not only what will be done, but how well.
- Early and frequent iteration, early feedback with the user by early delivery.
- Completion of analysis, design, build and test in one step.
- User oriented approach, the requirements are redefined after each delivery.
- Open-ended systems architecture with strong aspects on non-functional requirements.

At this point in the education, the students have gained an appreciation for quality management. In the pre-project phase they develop a quality plan for the project, starting from a quality standard in accordance to the principles of ISO9000 [9]. The quality plan defines procedures for configuration management, quality assurance and the various

kinds of documentation. Quality assurance is primarily based on inspection techniques.

The project leader in this course is an external software consultant. As before, the customer roles are played by teachers. Other roles in the organization defined in the quality plan like subproject leader, configuration manager, delivery manager, documentation manager, inspection leader, etc., are all played by students.

4. Experiences

In the practice driven courses it turned out the amount of work the students had to put in to complete their hands-on assignment was noticeably increased compared to conventional teaching. During this time the students had no additional teacher guidance. We found that giving the students the freedom to make and learn from their own mistakes provided them with sufficient motivation to make this extra effort.

The students were enthusiastic about the role playing approach used in the project courses. It turned out that as early as in the individual project, the students needed very little technical guidance and support. They were able to find answers to their problems in manuals, literature or sometimes from the tool suppliers. We learned not to underestimate the technical abilities of the students. On the other hand, the students found out that the deceivingly simple problem of making and keeping promises hides many difficulties to be mastered. Negotiating with a customer was a new experience to most students and in the first attempts they generally accepted too much work. They quickly learned the rules of the game, however, and soon started to explore the limits of their power.

We learned that it was vital that the teachers play their roles distinctly and clearly demonstrate which "hat they have on". The teachers often knew the technical problems "too well" and had to be careful not to include any advice on the solution when stating their needs as a customer.

The lack of project management lectures in the group project course created some initial difficulties for the students in dividing the work among themselves and at the same time maintaining sufficient communication about the various parts of the system. From the beginning,

though, they acted professionally towards the customer and concealed all internal conflicts.

Running the course in Project Organization and Human Work Science in parallel with the Group Project turned out to be successful with mutual benefits to both courses. Recently having had concrete experiences, the students were strongly motivated for discussing problems of human interaction.

The idea of rotating the groups for the modification part of the Group Project turned out very well. Although sometimes quite severe criticism was raised about documentation quality, code structure and design choices, the discussions were constructive without any bad feelings.

Grading project courses is a difficult issue. We grade a group as a unit as either pass or fail, where the grading criteria is the performance from the customers point of view. Failures on a project course are very rare since the customers have the right to claim a compensation in the form of extra functionality when the final result can not be accepted. When this "fine" has been fully paid, the group has passed the course.

The large project course this first year was very much a learning experience for the teachers as well as the students. We learned, for example, that we need to improve the project management. Having an external professional as project manager was very valuable, but his limited presence at the university created some difficult communication problems. Next year, we intend to separate the responsibility into two roles, with an external person still playing the role of the official project manager but with a permanent teacher as an assistant project manager. To our satisfaction, we noted that the students eagerly accepted the need for formally defining the development process and its quality procedures.

When the large project started, the problem was less well defined than we had wanted. Although this caused quite a bit of confusion and inefficiency, it also gave a realistic illustration of real life development, where the developer must take active responsibility to ensure he is solving the right problem.

A serious issue is the grading of the project courses. So far we have graded a group as a unit, either pass or fail, and the main grading criteria has been the performance from the customer's point of view. Whether this is the right way to do it, we do not yet know.

Almost a year after the first students graduated we did a follow-up with their employers. The general comment was that our students needed less time of internal training to become productive compared to students who had gone to ordinary engineering schools for four and a half years. One employer explicitly said he preferred hiring people with this background. In particular, their project experience was highly valued. As new engineers they had an appreciation for the different roles that are needed in a development, so they could easily start taking responsibility as configuration manager, documentation manager or basic test manager. No employer could mention an important area in which the students had insufficiently training.

Being a teacher in practice driven education is both rewarding and demanding. It has given us great satisfaction to work with students that are active and eager to search for knowledge. It is also difficult because we are influenced by the conventional style of the teachers during our own education. Using a practice driven approach, the teachers must live as they preach and actively search new knowledge by themselves.

References

[1] F. P. Brooks, "*No Silver Bullet - Essence and Accidents of Software Engineering*", IEEE Computer, Vol. 20, No. 4, April 1987, pp 10-19

[2] P. S. Gang, "*Rethinking Education*", Dagaz Press, Atlanta, Georgia, 1989, p 70

[3] J. M. Carroll, "*The Nürnberg Funnel*", The MIT Press, Cambridge, Massachusetts, 1990

[4] H. Abelson, G. J. Sussman, J. Sussman, "*Structure and Interpretation of Computer Programs*", The MIT Press, Cambridge, Massachusetts, 1985

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "*Object-Oriented Modelling and Design*", Prentice-Hall, 1991

[6] J. P. Jacquot, J. Guyard, L. Boidot, "*Modelling Teamwork in an Academic Environment*", Proceedings of the 4th SEI Conference on Software Engineering Education, pp 110-122

[7] L. Ohlsson, C. Johansson, "*An Attempt to Teach Professionalism in Engineering Education*", World Conference On Engineering Education, Vol. 2, Portsmouth, September 1992, pp 319-324

[8] T. Gilb, "*Principles of Software Engineering Management*", Addison-Wesley, 1988

[9] ISO, "*ISO 9000 - Quality management and quality assurance standards*", International Organization of Standardization, 1987

PAPER II

Evaluation of Code Review Methods through Interviews and Experimentation

Martin Höst and Conny Johansson

Published in the Journal of Systems and Software, Vol. 52, Issue: 2-3, June 1, 2000, pp 113-120

Abstract

This paper presents the results of a study where the effects of introducing code reviews in an organisational unit have been evaluated. The study was performed in an ongoing commercial project, mainly through interviews with developers and an experiment where the effects of introducing code reviews were measured. Two different checklist based review methods have been evaluated. The objectives of the study are to analyse the effects of introducing code reviews in the organisational unit, and to compare the two methods. The results indicate that many of the faults that normally are found in later test phases or operation are instead found in code reviews, but no difference could be found between the two methods. The results of the study are considered positive, and the organisational unit has continued to work with code reviews.

1. Introduction

Introduction of software process improvement proposals requires careful investigation of which changes to choose. Software process improvement involves changing the currently used processes, but it is almost never certain that a proposed change actually will result in an improve-

ment. Instead of an improvement there may be no effect at all, or worse, it may result in that processes that previously were working as they should, are affected negatively. Evaluation of improvement proposals is an important task, which should be started as early as possible in the improvement process.

Evaluation can be performed in a number of different ways, such as interviews with the staff, controlled experiments, case studies, and pilot projects. In this paper, it is described how two different company internal code review methods, both with high focus on checklists, have been evaluated through interviews and experiment. The objectives of the study are to analyse the effects of introducing code reviews and to compare the two methods. The study has been performed early in the improvement cycle, and further usage of code reviews is based on its results.

The intention of this paper is to present both the results of the study and the design of the study. The presentation is intended to be detailed enough to make it possible to repeat the study.

In Section 2, the organisation and the change proposal are presented. The study is described in Section 3, and the results of it are presented in Section 4. In Section 5, it is shortly described how the organisation has continued to work with code reviews after the study, and in Section 6 some conclusions are presented.

2. Background

2.1 Organisation

Ericsson Software Technology AB (EPK) is divided into business centres where the development of software products is made, product management centres (called nodes) who put requirements on the products and are the primary customer interface, and research and corporate functions. Each business center is divided into several business units, which normally handle one (sometimes two or three) product each. EPK employs approximately 1500 persons in total, and each business unit has approximately the size of 20-30 persons. EPK develops software for a wide range of applications, including several types of telephone exchanges, base stations and mobile phone management systems.

The presented study was carried out at a business unit developing gateways between mobile networks and billing systems.

2.2 Change proposal

In the initial process, the developed code is tested in three steps:

- Basic test (BT) - this test phase verifies that the smallest building blocks are correctly implemented. Code coverage test, memory release check and limit tests are examples of tests performed. BT has white-box test characteristics.
- Function test (FT) - this test focuses on integration and use case test of the smallest building blocks that were tested in BT. FT follows the black-box test principle.
- System test (ST) - this test verifies that the system meets the requirements. Installation, security and performance tests are examples of tests performed.

Before the change, limited or no code reviews were performed before BT. The main objective of the study is to investigate if code reviews before BT improve the quality, and fewer faults are found in test and operation. There are two different code review methods that should be analysed, method 1 and method 2. Method 1 consists of general checklist items, while method 2 is on a more detailed level. The checklist items in method 2 are focused on specific issues which have been experienced as important for the business unit, and focuses more on performance issues and language specific items. Examples of checklist items from the two methods are:

- Method 1: Is the program divided into suitable parts for readability and maintenance?
- Method 2: Is appropriate memory management used, i. e. new vs. static memory?

2.3 Incentives for running the study

Good incentives and high motivation are important when running studies in an industrial environment. These are the primary factors for this study:

- Root cause analysis on faults found in operation has been performed during several years in order to find out the causes for the faults in operation. This process defines a step-wise sequence for how to evaluate, for example, in which verification or test phase the faults should have been detected. The analysis indicates that approximately 25% of the faults found in operation should have been detected in code review. This gives a good indication that code reviews would be beneficial to the developers in the business unit. It also motivates upper management to implement improvements concerning code reviews.
- Calculations have identified correlation between faults found in early test phases and faults found in operation. The number of faults in operation decreases if the number of faults found in the early verification phases increases.
- The study was performed at the suitable time in the project's time schedule.
- The study could be combined with an educational workshop where the participants could be taught methods for code review.

The participants had good incentives, from both measurements and intuition, that code reviews would decrease the amount of faults found in later phases. This was a prerequisite for doing the study.

3. The Study

3.1 Introduction

The two major evaluation objectives of the study are:

- Analyse the effects of introducing code reviews in general in the organisation. That is, investigate if fewer faults are found in test phases and operation if code is reviewed.

- Compare the two code review methods.

In order to meet the above objectives, the study has been performed in three different ways, which are described in Table 1 and below.

Table 1. Study objectives and evaluation means.

Objective	**Evaluate code reviews in general**	**Compare the two methods**
Evaluation means	Impact analysis Experiment Measurements from test phases after code reviews	Impact analysis Experiment

In interviews according to the impact analysis method, developers were asked about their opinion about the impact of the review methods on their work. They were interviewed for both methods, and the interview results can be used both to evaluate code reviews in general, and to compare the two methods. How the interviews were performed is described in Section 3.2, and the results of them are presented in Section 4.1.

The experiment has been designed in order to make a valid comparison of the two methods. The measurements that were performed in the experiment can, however, also be used to evaluate reviews in general, without considering that the methods are different. The experiment is described in Section 3.3 and Section 3.4, and the results of it are presented in Section 4.2.

Measurements from the test phases after the code reviews can be used to evaluate code reviews in general. The results of the measurements are presented in Section 4.3.

That is, the study has mainly been performed through interviews and experimentation, but also through measurements in test phases after the reviews.

In general, interviews can be performed very early when a process change proposal is analysed. It is often natural to precede an experiment with interviews, and then base the experiment on the results of the interviews. In this study however, the interviews and the experiment were performed during the same workshop as described in Section 3.4. This approach was chosen because the risk of introducing code reviews was considered low, the participants wanted to learn about code

reviews, and it was possible to combine the study with the educational workshop.

3.2 Impact analysis through interviews

If a process change proposal should be analysed before the changed process is actually used in development, this could be carried out by interviewing experts from different sub-processes, such as requirements specification, design, test, etc. The interviewed people should answer questions about their opinion of the effects of the change on their work in their sub-processes. This means that a number of people are interviewed for every sub-process. Their answers can be combined, and estimations derived for every sub-process. When this is done, estimates from the different sub-processes can be combined into estimates for the whole process.

The general process of performing impact analysis through interviews is described in [1] [2] where a number of alternative methods for asking questions and combining answers are described and compared. It is found that a promising interview technique is to ask the interviewed people to give their quantitative answers according to a triangular distribution (i.e. to give a lowest possible value, a most likely value, and a highest possible value) and then combine the answers according to an average distribution. This method has been used in the study presented in this paper.

All participants in the study were interviewed before the reviews were carried out. Since the interviewed people have experience from more than one sub-process, they were interviewed concerning more than one sub-process. Every person was interviewed once for each review method. Examples of questions are:

- Q1-Q4: How many of the faults that normally are found in [BT, FT, ST, operation] should be found in code reviews?
- Q5: What is the relative time it will take to perform BT if the code is reviewed compared to if it is not reviewed?

For every question the following metrics have been computed by combining the individual answers:

- The mean value (e).

- The standard deviation (s)
- c^2, determined as $c^2 = s^2 / e^2$, i.e. a normalised variance.

The standard deviation and c^2 may be used as measures of the uncertainty of the experts (high values correspond to high uncertainty).

When Q1-Q5 were analysed, answers from experts who have been involved in the test phase in question at least once were used.

3.3 Experimental study

The review methods have been evaluated in an experiment where people in the organisation actually carried out code reviews, and the result was measured. The experiment has been designed [3] to compare the two methods, but the data collected during the reviews can also be used to evaluate code reviews in general, without considering that the methods are different.

The basic intention was that code should be sampled before BT. In the study, some basic tests were, however, performed before the code was sampled. This means that some of the faults in the code already were found and removed before the reviews. The reason for this is that it is nearly impossible to exactly match a study like this, with all the participants' time schedules. The tests were, however, performed informally and not completely. The code was compiled before it was reviewed.

3.4 Programme of educational workshop

The study was performed within an educational workshop, where interviews were carried out and two C++ code parts (*c1* and *c2*) were reviewed. 8 persons participated in the study. They have experience from the organisation and are representative of the persons that actually would perform reviews, if reviews are introduced. The participants have either a Software Engineering or a Computer Science degree from a Swedish university. Two of the participants were newly employed and had limited experience from the work at the business unit. The other participants have 1-6 years of experience in developing software and some of them have experience from several of the different test phases.

The persons that performed the reviews are the same persons as those who were interviewed.

The reviewers have written part of the code. Our judgement is that this does not disturb the study too much, and faults found by the author of the code are included in the study.

The programme of the educational workshop has been developed with the objective to meet a number of requirements, both requirements concerning the education and requirements concerning the validity of the evaluation:

- Everyone should use both review methods and review the same code. This is mainly because everyone should receive the same education during the workshop, but also because it means that more data could be collected.
- Everyone should not start with the same review method. This is because people may be more motivated at the first review and interview than at the second. If everyone started with the same review method the comparison of the two methods would be biased by the order.
- The two review methods should not only be used on different code. If one review method is used on code with fewer faults this would bias the evaluation.
- For obvious reasons, no one should review the same code twice.
- A meeting where every fault is discussed should directly follow every review. This is both because it is a part of the education, and because it is important to determine exactly which faults that have been found in the code.

Some additional requirements were that every review should be preceded by an educational session, and that the interviews with respect to the methods should start directly after the educational sessions.

The programme of the educational workshop is outlined below:

- Introduction: General introduction of the workshop. Two groups are formed, group 1 and group 2. Both groups consist of 4 persons.

- Education 1: Group 1 is educated in method 1, and group 2 is educated in method 2. The education is performed through self (and team) studies.
- Interview 1: The participants are interviewed concerning the anticipated effect of using the review method they just have learnt. The interviews are performed in parallel with review 1.
- Review 1: People in group 1 use method 1 in individual review of code *c1*. People in group 2 use method 2 in individual review of code *c1*. All major faults are reported according to a fault report template.
- Summary meeting 1: The objective of this meeting is to reach consensus concerning which faults that are true faults and not false positives, and where they would be found if code reviews were not conducted. False positives are not considered further in the study.
- Education 2: Group 1 is educated in method 2, and group 2 is educated in method 1. The education is performed in the same way as education 1.
- Interview 2: The participants are interviewed concerning the anticipated effect of using the review method they just have learnt.
- Review 2: People in group 1 use method 2 in individual review of code *c2*. People in group 2 use method 1 in individual review of code *c2*.
- Summary meeting 2: All faults that are found in review 2 are discussed.

The educational workshop lasted for three days, and all interviews were performed during these three days. In this type of study, where the participants perform much of the work by themselves, it is important to provide coaching and encouragement. It is also important that all participants have the possibility to allocate enough time for the study.

3.5 Validity of evaluation

The study has been designed with the objective to obtain high validity (see for example [4] [5]). Threats to internal validity are threats that can bias the measured variables and therefore lead to false conclusions. Examples of threats to internal validity of a study are that the history affects the study, i.e. different compared methods etc. are used in different occasions, and that people's experience may not be the same for two compared methods. The intention is that the design should make the internal threats small. One threat is, however, that part of the code is written by the reviewers, although this threat is not considered serious.

The study is based on few samples due to that the number of participants is limited. This may mean that there is another threat because the study is vulnerable to the effects of people leaving the study. There are, as described later, two persons who have left the study. The threat is, however, not considered too serious in this study. Few samples also means that it is harder to find statistically significant differences in the collected data.

Threats to the external validity limit our ability to generalise the results. Two common threats to external validity are:

- Selection: This is the effect of performing the study with a population not representing industrial practice. This effect is not considered critical, since the study is performed with professional engineers.
- Setting: This is the effect of performing the study in a setting not representing industrial practice. Since the study is performed in an industrial project and the reviewed code will be part of delivered products, this effect is not considered critical.

That is, the study is performed in a real environment, with professional engineers in a real project.

4. Results of Study

4.1 Results from interviews

The results of the computations with respect to Q1-Q4 are displayed graphically in Figure 1, where the values with respect to method 1 are plotted. There is no major difference between the answers for method 1 and method 2. Therefore, the values with respect to method 2 are not presented. The curve in the figure represents the mean values of the estimates as described in Section 3.2. The standard deviation is displayed in the figure through an upper bound (mean value + standard deviation) and a lower bound (mean value - standard deviation). In the figure, c^2 is also displayed for the different phases.

It can be seen that the nearer the phase is to the code review, the larger fraction of the faults is expected to be found in the reviews. The absolute size of the standard deviation is about the same for all phases. It can, however, be seen that the uncertainty in terms of c^2, i.e. the normalised variance, is higher the longer from the code review the phase is.

According to the interviewed people, about 80% of the effort would be required in BT if the code is reviewed before, compared to if it is not reviewed before.

The subjective opinions of the interviewed people show that code reviews would mean that a large number of the faults would be found earlier. For the operation phase, the result can be compared to the results from root cause analysis (see Section 2.3), which suggest that 25% of the faults could be found in code review.

4.2 Results from experiment

Collected data

The faults that were found in the reviews are summarised for every reviewer in Table 2.

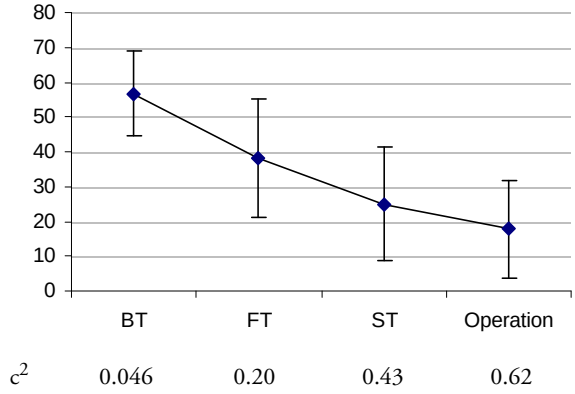

Figure 1. *Percentage of the faults that could be found in code reviews as judged by the participants.*

Every fault has been given a unique identity, e.g. 101, 102a. The faults that are distinguished by a letter (e.g. faults 102a and 102b) are faults with similar appearance at different locations in the code, e.g. "copy-paste" faults. These faults are treated as separate faults. It has been decided to do so because some of them have been found without that the other have been found by one reviewer.

In Table 2, 'Coverage' denotes how much of the code that was reviewed by the reviewer as judged by the reviewer. In the column marked 'Used' it is noted if the faults that have been found by the reviewer are included in the study. If the coverage for a reviewer is too low the faults found only by that reviewer are not included. Two reviewers had too low coverage.

Difference between the review methods

In this investigation the size of the reviewed code was 3.80 KLOC in review 1, and 4.11 KLOC in review 2.

In order to test if there is a significant difference, a paired t-test [6] can be used to test the null-hypothesis

H_0: mean(d) = 0, where $d_i = f_{i1} - f_{i2}$, and f_{im} is the number of faults/ KLOC found by reviewer i (1-5, 7) with review method m (1-2).

It is not possible to reject H_0 with either a t-test (p=0.65) or a Wilcoxon signed rank test [7] (p=0.44). The difference between the number of faults that were found with the two methods is small. It is not

Table 2. People participating in reviews

Reviewer	Review	Review method	Coverage	Faults found	Used
1	review 1	1	100%	101, 102a, 103	yes
	review 2	2	100%	209	
2	review 1	1	75%		yes
	review 2	2	100%		
3	review 1	1	100%	102a, 102b, 102c, 102d, 102e, 107a, 107b, 108, 109	yes
	review 2	2	100%	202, 205, 206	
4	review 1	1	100%		yes
	review 2	2	100%		
5	review 1	2	100%	102a, 102b, 102c, 102d, 102e, 110, 111,	yes
	review 2	1	100%	201, 202, 203, 204	
6	review 1	2	100%	105, 106	no (low coverage)
	review 2	1	15%		
7	review 1	2	100%	104a, 104b, 104c	yes
	review 2	1	100%	207, 208	
8	review 1	2	<50%		no (low coverage)
	review 2	1	0%		

possible to quantitatively decide which method to use in the future, and it seems like there is no practical difference between the methods. One explanation could be that the checklists were not fully used during the whole review. Even if the results of an investigation would be clearer, it is in many cases better to introduce small changes instead of larger in a first step. The risks of introducing too large changes may be too large. This means that it may be hard to find a significant difference.

A large number of faults were found by only one reviewer. This means either that the coverage of the reviews was low (and that different people reviewed different parts or aspects of the code) or that many

faults in the code were not found. The total number of faults in the code can be estimated with for example Capture-Recapture techniques [8] based on Maximum-Likelihood estimation. This analysis results in an estimate of 44 faults. The impression after the reviews was, however, not that there were many faults left in the code. The total number of faults in the code is further discussed below.

Evaluation of reviews in general

One objective of the evaluation was to estimate the effects of reviews on later phases of the project. For every fault that was found, it was estimated at the summary meeting in which phase (BT, FT, ST, operation) the fault would be found if no reviews were conducted. In some cases it was not possible to determine exactly where the fault would be found, and the reviewers had different opinions and did not agree on where it would appear. Then, an optimistic estimate is that it would be found in the earliest alternative and a pessimistic estimate is that it would be found in the latest alternative. The optimistic alternative is called optimistic because it is less expensive to find a fault early than late. For some faults it was not possible for the reviewers to estimate when the faults would appear if they were not removed. The result is illustrated in Figure 2. The actual curve is probably somewhere between the optimistic and the pessimistic curve.

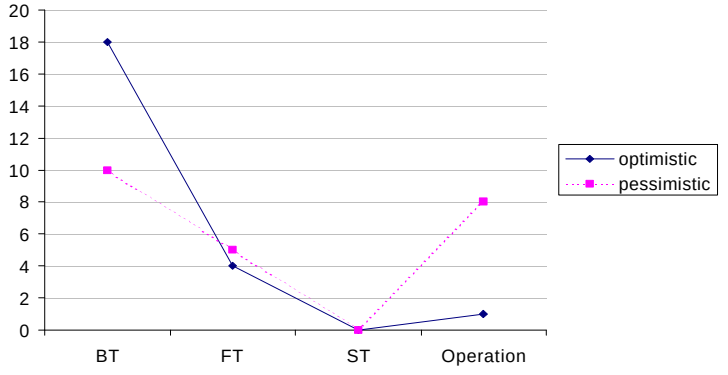

Figure 2. *Number of faults that would be found in each phase.*

The data in Figure 2 can be compared to experience data from previous similar projects. In Table 3 it is shown how many faults that have been found in different phases in previously developed code[1].

Table 3. Experience data.

Product	KLOC	FT faults	ST faults	Operation faults
Product release 1	117	37	18	2
Product release 2	78	45	28	4
Product release 3	115	3	54	5
sum	310	85	100	11

In average there are 0.274 faults/KLOC in FT, 0.323 faults/KLOC in ST and 0.035 faults/KLOC in operation (there are no measurements from BT). The reviewed code consists in total of 7.91 KLOC, and the expected number of faults has been calculated as:

expected number of faults = average number of faults per KLOC * size in KLOC

According to the experience there would be 2.2 faults in FT, 2.6 faults in ST and 0.28 faults in operation if code reviews were not conducted. This can be compared to the data in Figure 2, and it can be seen that the number of faults that were found in reviews is high compared to the experience. This may mean that many of the faults have been found, but the capture recapture analysis indicates that some faults remain in the code. The number of remaining faults is further discussed in Section 4.3.

The most often anticipated consequence is that the fault would result in a faulty function. For some of the faults, consequences like 'no visible consequence' or 'conflict with methodology' were anticipated. It could be argued that some faults would never result in a visible failure, but all faults included in the study have been reported as major faults. Minor faults were reported directly to the author, and not included in the study.

1. The code described in Table 3 consists of about 30% automatically generated code, but in the comparison the generated code is treated as regular code

4.3 Measurements from test phases

No additional faults have been found in FT or ST in the reviewed code. This may mean that most of the faults were detected in the reviews, and only few faults remain in the code. The reasons that the capture recapture analysis, as described in Section 4.2, indicates more faults can be discussed. One explanation is that each reviewer reviewed a large amount of code. This may mean that every reviewer did not review every line of the code with respect to every checklist item of the checklist, even if they reported a coverage of 100%.

5. Continued Reviews

Even if quantitative data are important as a basis, another important aspect is the participants' opinions after the reviews. The result of the study was presented at a seminar four months after the workshop. Their impression at the seminar was that code reviews are important, not at least since no faults have been found in the reviewed code in test. When the result of the interviews was shown, the participants agreed with the result and did not want to change their estimates.

The participants have continued to review code after the study. The total product consists of 215 KLOC. Approximately 25 KLOC are new developed code in this particular release. All new code has been reviewed, and the participants estimate a 20 to 30 percent decrease of faults found in function test and system test. The participants still have a positive attitude to code reviews (e.g., they are very interested in discussing the results of the study). The participants are now actively asking other team members to plan a review of their code, and they have continued to hold summary meetings after the code has been reviewed. They have refined the review process by introducing that the person who wrote the code conducts a walk-through before the actual review. This indicates that the participants have an interest in the process of code reviews.

The number of faults that were found in system test of the complete system was higher than expected. This is due to a lot of faults generated from system integration. The participants added checklist items in the code review checklist in order to overcome this problem. Check list items concerning the interfaces between the different sub-systems, and the testability and system utilization of the overall system were added.

This work may imply that the checklist is more important than initially estimated.

6. Conclusions

There is no large difference between the two code review methods. This is shown both in the interviews and in the experiment. In order to improve the study, we would use code review methods with greater difference. The checklists were too similar.

The developers' subjective opinion is that a large number of faults that normally are found in test and operation could be found in code reviews. Faults in early test phases that are near to the coding phase are more affected than later faults, but the importance of finding operation faults in code reviews are, obviously, higher. When the faults that are found in the experiment are analysed, it is found that many of them would be found in late test phases or in operation if code reviews were not conducted.

A large portion of faults has been found by only one reviewer. This may mean that different reviewers reviewed different parts or aspects of the code. Every reviewer reviewed a large amount of code during the three days. This may mean that too little time was spent on every detail of the code and that this is the reason that many faults were found by only one reviewer.

The study is considered successful. The participants found many faults during the reviews that they thought were important to correct at an early stage in the development. Their opinion is that many of the faults otherwise would be found later to a higher cost. Code reviews are accepted by the participants as a method for finding faults early in the development process. Through inquiries we have got the answers from the participants that code reviews are good for reducing the number of faults in operation, and it is better to carry out code reviews than try to cover all test cases with test instructions. The work with code reviews has continued in the organisation.

Acknowledgements

The authors would like to express their gratitude to the people who participated in the study with an open mind and a positive attitude. This work was partly funded by The Swedish National Board for Industrial and Technical Development (NUTEK), grant 1K1P-97-09673.

References

[1] M. Höst, C. Wohlin "*A Subjective Effort Estimation Experiment*", Information and Software Technology, Vol. 39, No. 11, 1997, pp 755-762

[2] M. Höst, C. Wohlin, "*An Experimental Study of Individual Subjective Effort Estimations and Combinations of the Estimates*", Proceedings of 20th International Conference on Software Engineering, 1998, pp 332-339

[3] S. Pfleeger, "*Experimental Design and Analysis in Software Engineering*", Part 1-5. ACM Sigsoft, Software Engineering Notes, 1994-1995, Vol. 19, No. 4, pp 16-20, Vol. 20, No. 1, pp 22-26, Vol. 20, No. 2, pp 14-16, Vol. 20, No. 3, pp 13-15, Vol. 20, No. 4, pp 14-17

[4] C. Robson, "*Real World Research*", Blackwell Publishers, 1993

[5] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, M. Zelkowitz, "*The Empirical Investigation of Perspective-Based Reading*", Empirical Software Engineering, Vol. 1, No. 2, 1996, pp 133-164

[6] D. Montgomery, "*Design and Analysis of Experiments*", John Wiley & Sons, third edition, 1991

[7] S. Siegel, N.J. Castellan, "*Nonparametric Statistics for the Behavioral Science*", McGraw-Hill International Editions, second edition, 1988

[8] S.G. Eick, C.R. Loader, M.D. Long, L.G. Votta, S.V. Wiel, '*Estimating Software Fault Content Before Coding*', Proceedings 14th International Conference on Software Engineering, 1992, pp 59-65

PAPER III

Software Engineering Across Boundaries - Student Project in Distributed Collaboration

Conny Johansson, Yvonne Dittrich and Antti Juustila

Published in IEEE Transactions on Professional Communication, Vol. 42, No 4, December 1999, pp 286-296

Abstract

Geographically distributed software development projects have been made possible by rapid developments primarily within the data communication area. A number of companies recognize that distributed collaboration has great potential for the near future. This report describes the empirical study of a co-operative student project located at two different geographical sites. The project was carried out at two universities, one in Sweden and one in Finland. The initial goals were to give the students the opportunity to learn about the practical aspects of co-operation between two geographically separate institutions and to study specific problems anticipated by the teachers with regard to communication, co-ordination, language, culture, requirements' handling, testing and bug fixing.

This report focuses on communication and co-ordination within the co-operative project as these were identified as the most significant problem areas. We also thought that these areas were the most interesting and the ones most likely to lead to improvements. This report not only describes our findings but also gives hints about what to think about when running similar projects both with respect to project-related issues and teaching issues.

1. Initiating Co-operation

Group projects are part of a number of undergraduate and postgraduate courses in software engineering and computer science. Simulating a realistic, industrial software engineering environment was identified as an important characteristic of successful project courses [1]. We did not do a great deal of research on other studies in distributed collaboration, though we think that this field is a neglected one within the software community. Selected distributed collaboration projects are described in [2] and [3], and specific research groups exist for example at the University of Hawaii [4] and the University of Illinois at Urbana-Champaign including their research partners. This paper does not present a detailed study of co-operating work; it is instead a personal record of the authors experience of one particular project run in a simulated industrial environment.

The initial planning of our joint project began seven months before the latter was actually launched. We started with a face-to-face meeting between the teachers involved at both geographical sites. During the initial meeting we outlined a provisional organization plan which was designed to serve as a guideline for the supervisors at both sites. We conceptualized the relations between the two projects as a sub-contractor relationship. It was decided that the main project should be located in Ronneby, at the University of Karlskrona/Ronneby, Sweden. The latter has approximately 3,000 students. The students at Oulu University, Finland were appointed as sub-contractors to the main project in Ronneby. Oulu University has approximately 12,000 students. The distance between the two towns is approximately 700 miles. Ronneby was selected as the main contractor due to the University's experience of large software projects. We thought that the project would be more manageable when one of the teams - the one with the greatest number of students - leads operations, with the other acting as the subcontractor.

As the client was to be a Swedish company, it was decided that the contract and the agreement covering the requirements' specification should be the responsibility of Ronneby. It was also decided that Ronneby should be responsible for the overall design and for dividing into sub-systems, preliminary estimations of the Oulu contribution, and the integration and testing of the final system. It was decided that the Oulu team should take the responsibility for the estimation and management

of their own work, complete the requirements' specification for their contribution, deliver the latter and correct reported errors. The requirements' specification and the partial deliveries were considered to be the most important product-related interface between the two teams. It was thought that problems related to organizational issues, process-related interfaces, problem and progress reporting, delivery dates and milestones, as well as defect reporting and correcting routines were likely to be activities which would connect the teams during the development of the project. A general framework for handling all these issues was developed. The concrete specification of the co-operation was left, however, to the students themselves. There were two reasons for this: as teachers, we wanted the students to take responsibility for managing the project; and as researchers we did not wish to restrict the students too much as to alternative choices of action.

The initial primary goal of the study was to investigate on a practical level possibilities and difficulties within the following areas:

- communication - electronic and other distance communication – –between students, students and teachers, and between teachers in different locations;
- co-ordination and different management cultures;
- language, including different native and software engineering languages (e.g. differences in definition and understanding of terms and connotations);
- requirement co-ordination;
- testing, focusing on integration and system testing;
- bug fixing.

In order to assess the fulfillment of the above goals, three students were attached to the project: two completing their Bachelor degree in Ronneby [5] and a master's student in Oulu [6], for whom the project provided empirical data for his dissertation. Their role was to observe the ongoing work, focusing on the communication processes both between the two groups, and between each group. In order to understand the evolution of the different concepts at both places as well as the difficulties in communicating these, ethnographic methods were applied and the results were related to the latest findings on software development communication.

Each of the three students stayed with the groups they were observing for the entire project period. In order to prevent the students influencing the communication process by providing additional information, it was not permitted to discuss project observations during the period of observation.

At an early stage, some two months before launching the project, we decided on the customer and the subject for the project: a business simulator for use in management training. The co-operating company is one of Sweden's largest independent information technology service suppliers. It has around 750 employees and branches in approximately 20 towns in Sweden, Norway and Denmark. The company has a separate division responsible for simulation and training systems. The project comprised not only the engine for simulation but also the creation of the scenarios for simulation as well as the simulation evaluation. The Oulu team was contracted for the evaluation application used in the graphical presentation of the data logged during simulation.

2. The Project Courses: Aiming at Realistic Settings

As both projects were part of already existing project courses at the two universities, an important pre-requisite for co-operation was the maintenance of the basic principles governing the courses. Both courses - the large team project in Ronneby, and the programming project in Oulu - aim to provide a realistic setting for the students. This is primarily achieved by co-operating with an industrial partner who adopts the role of a real client. The students gain important practical experience which is closely related to their academic studies, with the additional advantage that the environment is controlled. In Ronneby, all project courses (which comprise one year's study in total) are taught in co-operation with industry in order to provide the students with realistic settings where they can apply and practise their skills. The two projects differ in other important aspects, however. The main difference between the Ronneby and the Oulu courses is in the group sizes and ranges. The Ronneby project comprises all the stages from negotiation of customer requirements to final acceptance. Oulu focuses to a greater extent on handling all stages from the initial design to the final testing. In Oulu, the requirements' phase is given less attention than in Ronneby.

In the following the project team at the University of Karlskrona/ Ronneby is called 'the Ronneby team', and the project team in Oulu is called 'the Oulu team'.

2.1 The Ronneby Project Course

The 'Large Team Software Engineering Course' at the Department of Software Engineering and Computer Science at the University of Karlskrona/Ronneby covers 20 weeks. One week of study comprises 40 hours of lectures and individual study. Third year students on the undergraduate Software Engineering Program are divided into teams of between 12 and 20 students.

The course has been revised several times during the eight years it has been running. Two to three projects are started each January. During the period of co-operation with Oulu the project was staffed by 14 Software Engineering students: three from the 'People, Computers and Work Program' (students study 50% computer science and 50% human work science); and two from the Business Administration Program. The 'People, Computer and Work' students were mainly concerned with how people might use the software at work, while the Business Administration students focused on project costs. The non-Software Engineering students participated in the project as part of the course requirements of their own program of studies.

The Large Team Software Engineering course does not only focus on the programming element but also on pre-study, management and quality planning issues. During the pre-study phase, which constitutes approximately one third of the project, the project is defined and the customer requirements are identified. Quality planning includes specification of guidelines and rules to be applied to the project. The goal is for the product to conform to the relevant ISO standard. Project planning and progress and cost tracking are important cornerstones in every project and are given high priority in the course. Each student is allocated a budget of approximately 700 hours, with a total project budget of between 8,500 and 16,000 hours. All roles in the project are assigned to students with the exception of the Head of Department and Quality Manager; these are allocated to teachers. The development model used is based on evolutionary delivery [7]. Strong emphasis is put on inspections and reviews, defining and tracking non-functional requirements, and handling complex situations and problems common in large

projects. The Large Team course, including the roles mentioned above, is described in detail in [8]. Past projects include:

- a mobile positioning system using the GSM (the Global System for Mobile communications) net;
- a library system;
- an integrated system for administration and billing of cellular phone calls and services.

In order to understand the large number of problems within the Software Engineering area it is extremely important that the students first gain experience of more minor tasks and projects. The aim of the individual and the small team project which preceded the Large Team project was to introduce tricky areas such as client relations, requirements specifications, project planning and tracking, along with customer and team commitment culture.

2.2 The Oulu Project Course

The Programming Project Course at the Department of Information Processing Science at Oulu University is of five weeks duration and brings together courses covered in the first two years of study. By the end of these courses students acquire theoretical knowledge and limited practical experience of programming, database design and implementation, project management and software engineering. The aim of the project course is to put this knowledge into practice. This is done in as realistic a manner as possible and in co-operation with software companies in the local area. The course is held every autumn and spring; 6 to 9 projects normally start in the autumn, and 1 to 3 in the spring. In addition to working in local companies, there is usually one research project per term, the aim of which is to produce and test new technical ideas. These projects normally involve 4 participants, each of whom is required to work approximately 200 hours.

On completion of the project, the student is expected to apply his/her theoretical knowledge of software engineering to practical situations, to show an ability to work as a member of a team and to be able to communicate verbally and in writing with external and internal interest groups. Students should also have experience of designing, implementing and testing software. Past projects include:

- adding support for a data glove for software supporting distributed design over the Internet using the programming language Java and the file format VRML (Virtual Reality Modeling Language);
- adding EAN (European Article Numbering) code support for car sales software;
- improving software functionality in the analysis of software production metrics.

The course has been in co-operation with local companies for over ten years. This has proved to be a highly successful form of co-operation between the Department and companies involved. Students gain important practical experience which is closely related to their academic studies, with the additional advantage that the environment is controlled.

A software management project which is closely related to the programming project course is held each autumn and spring. An older student who has already completed the programming project is appointed project manager of a team. The project manager is responsible for planning and directing the project, supporting the team and for communication with the project control group. The Oulu Project Course is described in greater detail in [9]. During the following year of study students carry out a more extensive and advanced project comprising 250 hours. The following stages are covered: problem definition, requirements engineering, software architectural design, and software design and implementation.

3. Obstacles to Co-operation

The inability to co-operate and communicate effectively with other participants is one of the major barriers to the achievement of goals in software development projects. This weakness is also of course characteristic of other kinds of projects. Poor communication and co-operation between team members, management and team members, and even between managers have frustrated many companies in the past [10]. Spatial distribution of projects adds to these problems: direct face-to-face communication is not possible, misunderstandings easily arise and are detrimental to common action, and it is not possible to rely on facial

expression as an indication of poor understanding. The majority of the problems arising during the project were due to different software development cultures, inadequate co-ordination practices and communication. Those issues which had a direct bearing on our project are discussed below.

We have not attempted to describe the practical progress of the project but have instead concentrated on the most interesting findings from a communication point of view as well as on those findings which are directly related to the concept of distributed collaboration. The analysis is divided up into the following sections: power considerations, conflicting assumptions, communication and supporting tools, and language and cultural differences.

3.1 Power Considerations

The Ronneby team consisted of 19 students and the Oulu team of 3 students. The difference in group size and contractor-subcontractor relationship between the two projects resulted in an imbalance in power between the partners. On several occasions the Ronneby team decided to change direction. It discussed and 'solved' problems which also influenced the Oulu contribution without either consulting the Oulu team or allowing it to take part in the decision-making process.

The Oulu students interpreted the part of the sub-contractor too literally sometimes. They expected to be told exactly what to do: 'I think that they will define the database . . . that the database description will come from [Ronneby]' [6]. Both teams agreed that Ronneby took the initiative and had the final responsibility for the finished product. The Oulu team saw their responsibility as being limited to the fulfillment of orders received. This resulted in a lack of reflection on and response to the specifications received from Ronneby. Much time was lost at Oulu due to a lack of ability to design and implement in accordance with what were all too frequently insufficiently detailed specifications. Significant changes during the actual running of the project also made it more difficult to keep the agreed deadlines.

Language problems (these will be discussed in greater detail later) added to the general difficulties. The proposal request and requirements' specification were written in Swedish and were not translated into English (or Finnish). The Oulu team thus lacked detailed information.The issue was however, never raised. The teachers were unaware of

the problem until the middle of the project. The neglect of this aspect resulted in delays on the Oulu side as well as in general frustration in the Oulu team. Awareness of the problem resulted in a change in attitude. During the second half of the project the two partners were on a more equal footing.

It is important to remember that the participants were students. The latter lack routine and do not normally have the professional's ability to manage complex subjects without guidance and support. This was the first 'real' large project for the participating students. It is often difficult enough for the students to understand and cope with the knowledge gained from university projects. In the situation described in this paper, students must also understand the differences between the project paradigms at both the universities, and adjust their behavior accordingly. This is an important aspect for teachers to consider both before the start of the project and during the actual project period.

3.2 Taking Their Own Way of Doing Things for Granted

As mentioned earlier, the parties involved represented two different universities and thus two different software cultures. It is a necessary for teams to assimilate the local culture as well as the organizational culture of other teams, in order to be able to co-operate effectively [11]. The Ronneby team expected the Oulu team to understand that the requirement specification document should specify client requirements but no solutions i.e. design solutions. Oulu, on the other hand, expected the document to be complete and include at least some design description. It is standard practice in Oulu projects for requirements to be handed over to the teams at the same time as the preliminary design documents. The student team is responsible for finishing the design and starting the implementation of the software. As Oulu was designated the sub-contractor and Ronneby the main contractor, Oulu expected Ronneby to produce the requirements' specification. It is important even when playing the role of sub-contractor to participate in the requirements' phase and to investigate actively the possibility of fulfilling requirements. This is also possible, and is indeed even more necessary when the sub-contractor does not, as we did, have direct contact with the client.

An additional cultural aspect is management ethics. Throughout the program, the Ronneby team is trained to act as managers, with an emphasis on co-ordination and progress. A manager should not make decisions by himself and should delegate tasks to team members. In the project described here, Oulu had no project manager with whom Ronneby could negotiate. The Ronneby team were thus unsure as to who was responsible for communication and co-ordination. The emphasis should be placed on commitment culture [12]; that is, everybody should participate in the decision-making process, and that the decisions which are taken must at least be acceptable to the majority. This problem is also, of course, in part a function of the fact that the project was included in two different courses which were managed in different ways due to the different software management cultures. In Oulu, for example, reporting was considered a difficult task: 'how can Ronneby expect daily reports when they haven't given us enough information to start working on?' [6].

The definition of terms and concepts is extremely important i.e. specification of requirements, architectural design, unit and integrating testing. The Oulu course focused on design, programming and testing, while the Ronneby course covered the entire software development cycle up to actual operation, with a special emphasis on project planning and estimations. It is extremely important not to take for granted what appear to be obvious objectives and definitions. These must be clarified and, where necessary, changed.

The Ronneby team focused more heavily on documents than did the Oulu team. These documents included the requirements' specifications, test specifications, project plans and progress reports. The difference in focus between the two teams was the result of differences in culture and work routines. When the Ronneby team expected a complete document, for example, it sometimes only received a few phrases of instruction. The Oulu team, on the other hand, was overwhelmed by the volume of documentation produced by Ronneby and found it difficult to ascertain which documents were important as far as their responsibilities to Ronneby were concerned.

Documentation related to the development paradigm of Ronneby which followed the evolutionary process also created problems as Oulu found it difficult to understand why Ronneby filled the electronic work space for document exchange with documents which contained little more than titles and subtitles. The following quotation from the analy-

sis of the project illustrates the problem: 'it would be good if they had one single document from which one could see what is being done and not just a bunch of two-line explanations scattered between several documents' [6]. When new increments of the specifications were subsequently added the matter was clarified. The same thing happened with the database which formed the interface between the two parties. The database structure changed frequently. It was necessary for the Ronneby team to provide sample data. The Oulu team found it difficult to cope with the frequent changes.

3.3 Locality of Everyday Communication

Much of the everyday communication and co-ordination within software development takes place outside formal meetings and is primarily of a face-to-face nature. Coincidental meetings and mutual awareness of work tasks are thus of crucial importance. The common project room provided a good example. The students observing co-operation efforts used the project room a great deal to observe the group and their meetings as well as for their own work. These students became a more integral part of the project than those working on their own. Morning meetings were very short and only took place on rare occasions. The members of the core group knew what was going on, and the others did not know what to ask for. Weekly meetings and special purpose meetings were used for other issues. As the Oulu team was unable to participate in this local everyday communication it did not receive the information necessary for identifying difficulties at an early stage and keeping itself up-to-date as regards the work of others which could have some influence on its own particular tasks.

No one in the Ronneby team was conscious of this awareness problem. If such awareness were furthered it could lead to informal interactions, spontaneous connections and the development of shared cultures [13]. The need for information and co-ordination with the Oulu team was not recognized until the Oulu project was already somewhat behind schedule. It was subsequently decided that communication between the two groups should be a top priority. This was improved significantly with the aid of e-mail, which compensated to some extent for the missing face-to-face contact.

3.4 Groupware Does Not Guarantee Communication

The projects followed our recommendations and the BSCW (Basic Support for Co-operative Work) [14] was used, as a basis for sharing documents and co-ordinating work. BSCW uses the Web, thereby enabling access from all computers with internet facilities. It allows documents to be shared and for links to be formed with defined groups, as well as for awareness of events in the particular workspace (who has read, changed or up-loaded what and when) and version control. It also enables newsgroup-like discussions to take place. Communication thus takes place through the common artefact - the shared documents, and what happens to them - and it allows for communication about the common task as well as for explicit co-ordination [15].

At the beginning of the project, the Ronneby team decided to have open access for the Oulu project to all project documents located on a BSCW server. The work space and the documents themselves were organized in accordance with the particular needs of the Ronneby team. The specific needs of distributed collaboration were not taken into consideration.

The Oulu team was unsure about where to look for relevant information. It was not informed about the change in important documents and nobody seemed to read his/her own documents. The BSCW was not used in a way that was calculated to further distributed collaboration.

We as teachers suggested the BSCW be used. This suggestion was accepted by the students without further evaluation. Our experience of the project led us to have doubts about the practicability of using the BSCW for this form of co-operation. We are not, however, able to specify the exact nature of our criticism at this stage and have thus not focused on this in our research.

3.5 Language and Culture

We do not have the expertise to analyze whether the misunderstandings and difficulties that occurred were due to different native languages or cultures. During an analysis of a chat protocol, however, one important aspect cropped up, namely, that both sides were intent on keeping each other happy. This may be related to the Nordic tendency to avoid conflicts. According to Hofstede's [16] studies on national work-related

cultures, both the Swedish and Finnish cultures are individualistic. Swedes and Finns are anxious not to lose their self-respect or guilty. This may have been influential in the tendency to avoid conflicts. The Oulu team, for example, ignored requests for comments on documents because it did not feel sufficiently competent and did not wish to ask questions for fear of appearing stupid. Another example is provided in the following comment: 'The Department teacher who supervised the project established with the Oulu team at the beginning that it should keep sending unanswered questions back to Ronneby until it received the required information. It seems that the project group, even though they agreed to these tactics, never really did this during the project. When I [Petman] asked the Oulu group why they didn't do it more often, the reply was that they didn't believe that "aggressive" requests would have helped at all' [6].

An indication that there may have been hidden traps in language use showed up afterwards. During a conference attended by one of the authors, a Finnish researcher proposed that problems (regarding software development) should not be discussed in great detail. No one cares to admit he/she has problems; the use of such expressions as 'solving difficulties' and 'improving practice' are, however, acceptable. Further inquiry among Finnish-speaking colleagues showed that there are two possible Finnish translations of the English word 'problem': 'tehtävä', meaning 'a task to be solved'; and 'ongelma', meaning 'trouble' and suggesting the need to place blame somewhere. 'Tehtävä' describes a riddle or a puzzle. 'Ongelma' is used for more critical situations, when it is uncertain if the problem can be solved and where there is no identifiable remedy. It is possible that the Finns never reported 'problems' because they equated the word with 'ongelma'. Software engineering literature refers to 'problem definition' and 'problem decomposition' as a starting point for analysis and design. In teaching software engineering, the Finnish author of this paper finds it irritating to hear people talk on occasions in terms of 'defining a problem' when discussing how to start a design: A problem is not necessarily a pre-requisite for design work. Software may quite simply be used to achieve a faster or a better solution to a given task.

4. Lessons Learned

Overall the project must be regarded as a success. Results were delivered on time and the client was satisfied with the quality of the end product. Despite the difficulties involved in a co-operative project of this nature, the students generally felt the experience was useful and are in favor of a second trial. We will, however, make changes in the next project. As teachers we have learned a great deal about distributed collaboration in general, as well as how to supervise co-operative projects. The following sections offer a summary of the lessons learned as regards distributed collaboration in general. Pedagogical aspects are also discussed.

4.1 Caring for Co-operation

Communication and co-ordination are extremely important issues within software development [17]. Developing a common representation of the future application, decomposition and related distribution of work, co-ordination of the evolving design and the (re-)composition of the software under development are crucial for a project's success. They require a common ground for co-operation and communication, and the development of a commitment culture. A common methodology and technology combined with close co-operation often result in implicit agreements on the foundations for future co-operation. Practice and experience dictate which documents are to be reviewed by whom. When and how important decisions are to be taken, and whose perspective is to be taken into consideration are common knowledge. The person who is to be informed about the kind of changes to be made, and the one to ask for help need not be specified. This knowledge gradually emerges as a result of common experience.

The implicit nature of organization and co-ordination may lead to problems in projects in which there are as many as 15 to 20 participants. In our particular case, project members who were not consistently present missed important developments and decisions and were left behind. The problem of implicit expression is particularly acute in distributed collaboration and resulted in our case in major delays, as well as in an over-running of the budget.

Even where the basic rules of communication are taken for granted e.g. politeness, answering e-mails, criticizing in a constructive way etc.,

the specific problems of distributed software development may require very specific measures. As there is very little literature on communication as far as distributed collaboration is concerned, we looked for alternative areas with similar problems: in participatory software development, for example, co-operation between users and developers with different professional backgrounds must be maintained across organizational boundaries. The following measures, based on [17] and [18] will be tried out in a second co-operative project. Our aim is to adapt the basic ideas and techniques of participatory software development with respect to perspectives and roles, explicit foundations of co-operation, development of a common language, and mutual integration and information to the specific problems of distributed collaboration.

4.2 Establishing Common Grounds

Agreement on, maintenance of and renegotiation of commitments are fundamental principles of professional software development [12]. Commitments are based on agreements about what is to be done, who is to do it and when it is to be done. Important considerations also include identifying who is reliant on the task being completed, who has to be consulted about changes, and how to find necessary information as well as who possesses this information.

How a team actually arrives at an agreement is not always easy to assess: the absence of objections during a project meeting or of remarks about a management document may suggest implicit agreement in one organization, but disagreement in another. Explicit agreements are particularly necessary when it comes to co-operation across organizational, cultural and/or professional borders. A hierarchy between the partners may cause additional problems since agreements can only be made when it is possible to disagree. If one party is forced into a commitment, passive resistance may result. Differences in the degree of power allowed as a result of different forms and degrees of co-operation (contractor-subcontractor) must be given proper consideration.

What is agreed upon is an additional source of misunderstanding in distributed projects: what does the term 'progress report' mean, for example, and why is it important to the overall project? How a specific document is to be used may be common knowledge in one organization. In another, documents with the same name may be used in a

totally different way. Tasks implicitly related to specific roles, review habits etc. are additional sources of misunderstanding.

In our particular case, identification of project participants and the periods of active participation were not usually difficult to assess: people were easily identifiable and the different time zones (there is a one-hour time difference between Sweden and Finland) did not pose a problem. This could, however, be a problem which requires explication in projects where groups are involved whose understanding of the nature and importance of time is different. Frustrations and delays may be avoided by being aware of the existence of such problems, devising explicit descriptions or even compiling a management-related glossary with definitions which should be agreed upon rather than imposed.

4.3 Keeping the 'Distant' Group Present

Being able to act spontaneously when discussing issues which influence one's sub-task and to contribute relevant arguments that may be important for the project as a whole are important if complex systems are to be developed co-operatively. Grynter [19] shows the importance of co-ordination of changes and implementation throughout development of a system. Both formal meetings and spontaneous chats are used to keep participants up-to-date and to allow for individual contributions suggested by the participant's particular task [19] [20].

Despite advances in technology, video conferences are still problematical in the opinion of the present authors: transmission is usually slow, the setting is artificial, it is confined to a specific room, and taking turns is cumbersome. With the media itself in focus discussion is difficult. Video conferences may work in special cases but are not ideal for everyday communication.

We recommend that a virtual member be appointed on each side to represent the remote project. One team member can represent the remote sub-group at meetings, speak on its behalf, and inform about relevant issues. He or she should adopt the point of view of the remote team and remind the group when it is necessary to co-ordinate the task with the remote team. The representative needs to discuss the agenda in advance with the remote team and should be aware of what is going on in the other project. The task of the virtual remote team member is not, however, to control the remote group but to speak on its behalf.

Mutual integration and information can also be furthered by organizational means. Review cycles can be defined in such a way that each team reviews the documents relevant to its own tasks. Misunderstandings, implications of changes, errors in design decisions with regard to interaction between both parties may be identified early in this way. Quality assurance may be used as a method of keeping the teams informed about relevant developments on the other side and as a support for building up a mutual understanding of the common task.

Supporting spontaneous chats is a more difficult task. Even when the team members were permitted to chat or phone, they made little use of this opportunity. Chats - the most spontaneous computer-mediated form of communication - were used only three times, and then only for management purposes and not for the co-ordination of everyday work. It was originally intended that the phone be used more frequently in our project, and that all calls be recorded. It proved difficult, however, to gain access to a phone and a tape recorder. Bellotti and Bly [20] show how difficulties may occur in spontaneous co-operation even when the distance between the project sites is as little as 20 miles, and when there is no language barrier. They argue that lack of awareness lies at the root of the problem: the participants studied did not know if a person at the other site was sitting at his/her desk, they could not see what he or she was working on, neither was it possible to overhear discussions.

Naturally, communication technology cannot mediate all aspects of face-to-face communication. Certain technical solutions are, however, perfectly possible. Various ideas have been tried out and proved to be successful, including the abstract representation of noise level and movement in a remote room [21]; or web-based project sites where group members can 'hang out' in a virtual sense [20]. Low-level support such as a list of project members who are logged into the project account and who are working on their computer, or even a video picture from the room occupied by the remote team, can show who is present and who is available for a chat.

4.4 Developing Common Practice

The measures discussed above are intended as support for distributed collaboration. Co-operation is essential when there is a common task/ product. As a means of reducing risks, however, on the other hand loose links are advisable. A small interface only is in the general interest. In

our particular case, the links were too loose. The common task was not sufficient to encourage communication and co-operation. In our effort to reduce risks by depending on an unknown group, communication and co-operation suffered.

We are unable to offer any solution to this trade-off. We intend to experiment with a different division of tasks which are dependent on communication and co-operation. A possible example is the early development of a prototype of an important unit which must be further developed by the other team. This would encourage the project teams to adapt and to use electronic communication media in a way that makes sense to them. It would also encourage the development of communication routines based on the communication media available.

4.5 Teaching Distributed Collaboration

It is important that the software engineering teacher who aims to re-create a 'real' practical situation does not direct or steer too much. Understanding increases significantly if the students are allowed to experience difficulties as they occur naturally during the project. As a teacher you must have the courage to put yourself in uncertain situations in which the course of events is largely unpredictable. It is necessary to have an ongoing assessment of the degree of teacher involvement required. Sometimes more intervention is necessary, sometimes less. The following section describes some of the issues which require a greater degree of teacher involvement than that which characterized our project.

The sub-contractor and the supervisor at the sub-contractor site must be more active in both drawing attention to and keeping in touch with the main project, and in explaining the situation. The manager and supervisor at the main project location, on the other hand, must be more aware of the fact that the sub-contractor may lack information. One solution is to use simple checklists on which items are continually ticked off i.e. at formal meetings held throughout the duration of the project. Checklists should include points related to co-ordination and communication and should be used by both teachers and students. Co-operation between the teams involved in our project would undoubtedly have been considerably improved had these issues been raised at the weekly project meetings.

The teachers on both sides agreed that the students should be responsible for managing communication. An understanding of differ-

ent forms of communication and of methods of improving communication comes with practice. It is necessary, however, that teachers monitor communication to ensure that it functions adequately. There should clearly have been more informal contacts between the teachers in our particular project.

We decided against having management groups between the teachers at both locations. This may have been a mistake since there was clearly insufficient focus on joint efforts. There should have been more contact between the teachers involved, although allowance must be made at the same time for initiative on the students' part. It could be argued that teachers should have more power i.e. make decisions at the management group meetings. It is necessary to strike a balance between teacher influence and adequate opportunity for the students to gain by the experience of making mistakes.

5. Summary of Improvement Proposals

Before initiating any co-operative project over a large distance it is necessary to consider the nature of the co-operation aimed at, and the project to be carried out. Does the latter require close co-operation and communication in the specification of the elements to be developed jointly? Is it possible to isolate clearly separate components that may be contracted with minimal interfaces between the participants? What are the implications of this in assembling communication channels with regard to personnel responsibilities and to documentation guidelines? The following checklist may prove useful but must be adapted to the particular project in hand. We believe it provides a framework for important issues.

- Do not take your own definitions for granted. Be explicit as regards the meaning of terms and concepts. A common method of defining terms should be agreed upon. The most obvious example is to define what is meant by the term 'requirements' specification' both with respect to the actual activity as well as to the document.
- Give greater emphasis to the commitment ethic. Commitments must be openly stated and each party must emphasize its particular speciality. People are different, some work fast and some

slowly, and we all have different interests. All this should be openly stated without anyone feeling he or she is in some way inferior.

- More direct contact must be taken which is not specifically related to written information. The telephone, video conferences or chats are more useful than e-mail contact. Provide the technical means of providing at least low-level awareness of the distant group.
- Introduce the role of a virtual member of the distant group. The chosen student, who may perhaps work on the other side of the interface, should be responsible for communicating information and preparing decisions. The virtual member should work as a gate-keeper [22] and disseminate relevant information from the outside to project members. He/she should also leak information to outsiders. The virtual members should keep track of interface objects.
- The teachers involved should prioritize co-operation. A significant improvement may be gained by quite simply raising the issue of communication at meetings. Although communication will almost certainly not be given the highest priority (solving urgent client problems will be given higher priority), it is extremely important that teachers almost over-emphasize the importance of co-operation and good working communications between the two sites.
- Part of the teams should meet at least once in order to establish co-operation. An interesting experiment would be to exchange one member from each team. This might possibly solve some of the communication deficiencies e.g. when decisions are made at informal meetings.
- Try to create a good atmosphere by giving each other not only negative but also positive feedback. One of the most important things you can say is, 'Thank you very much'. In this way you inspire one another and create a good team spirit. Jokes are also good if these are kept at the correct level. Ice-hockey is an excellent subject for jokes between Swedes and Finns.

- If a co-operative system of support is used a distinct workspace should be allocated which contains only those objects which build the interface or serve communication between the teams. All other objects should be removed.

6. Conclusions and Future Plans

Overall it can be said that the co-operation project was definitely worth the effort: both the students and ourselves learned a great deal about distributed collaboration and about international co-operation. The problems the students encountered may be regarded as realistic experience. The accompanying research makes it possible to address a number of relevant research questions [23]. The project described in this paper enabled us to bring together research and basic education.

We were also able to develop new research ideas and plans: the rationale of the projects and their organization provided us with realistic and easily accessible empirical study options. During this first co-operative project the focus was very much on communication and the organizational side of co-operation. The material aspect - documents to be exchanged, the common product to be developed, integration, testing and debugging of the various versions - was not the main problem of the project and was thus not included in our research.

We intend when evaluating projects in the future to focus more on this particular problem area. The co-ordination around interfaces, common objects and documents is important in distributed collaboration. These can be regarded as boundary objects (site), fulfilling different purposes for each team but providing nonetheless a 'common ground' for co-operation. What kinds of patterns of co-operation [22], what kind of working distribution of labor [24] are developed around these boundary objects? How do the teams manage to integrate their contributions into the diverse deliveries and the finished product? What are the specific problems related to testing and debugging, and how are these solved by the teams?

The groupware we used, BSCW, did not stimulate the teams in sharing information. In future we will include studies and evaluations of other tools [11] [13].

As distributed collaboration is really no more than software engineering under very specific constraints, our results may highlight difficulties

in other projects which are otherwise hidden by developers who are skilled in getting round organizational pitfalls.

Acknowledgments

This paper could not have been written without the accompanying studies done by Monika Alriksson and Kari Rönkkö, students at the University of Karlskrona/Ronneby, and Timo Petman, student at the University of Oulu.

This co-operative work was initiated at the conference ISSEU '97, arranged by the Southampton Institute and Rovaniemi Institute of Technology.

References

[1] D. Gotterbarn, R. Riser, "*Real World Software Engineering: A Spiral Approach for a Project Oriented Course*", Proceedings of the 7th Conference on Software Engineering, 1994, pp 119-150

[2] P. Brereton, S. Lees, M. Gumbley, C. Boldyreff, S. Drummond, P. Layzell, L. Macaulay, R. Young, "*Distributed Group Working in Software Engineering Education*", Journal of Information and Software Technology, Vol. 40, Issue 4, 1998, pp. 221-227

[3] D. Bandow, "*Geographically Distributed Work groups and IT: A Case Study of Working relationships and IS Professionals*", Proceedings of the 1997 Conference on Computer Personnel Research, 1997, pp 87-92

[4] P. M. Johnson, "*A Proposal for CSDL2: A Center for Software Development Leadership through Learning*", Department of Information and Computer Sciences, University of Hawaii, Honolulu, Number ICS-TR-98-03, July, 1997

[5] M. Alriksson, K. Rönkkö, "*Softtalk. Communication in distributed software development*", Bachelor Thesis, Department of Computer Science and Business Administration. University of Karlskrona/Ronneby, 1998

[6] T. Petman, "*Communication Matters. Communication in Geographically Distributed Software Development Projects*" Master's Thesis. Department of Information Processing Science, University of Oulu, Finland, 1999

[7] T. Gilb, "*Principles of Software Engineering Management*", Addison Wesley, 1988

[8] L. Ohlsson, C. Johansson, "*A Practice Driven Approach to Software Engineering Education*", IEEE Transactions on Education, Vol 38, No. 5, 1995, pp 291-295

[9] A. Juustila, T. Molin-Juustila, "*Evolution of an Undergraduate Course in Software Engineering*", Proceedings of the ISSEU '97, pp. 101-108

[10] P. H. Jones, "*Handbook of Team Design*", McGraw-Hill, 1998

[11] R. Cutkosky, J. M. Tenenbaum, J. Glicksman, "*Madefast: an Exercise in Collaborative Engineering over the Internet*", Communications of the ACM39, 9 (Sept. 1996), pp 78-87

[12] W. S. Humphrey, "*Managing Technical People*", Addison-Wesley, 1997

[13] P. Dourish, S. Bly, "*Portholes: Supporting Awareness in a Distributed Work Group*", Conference Proceedings on Human Factors in Computing Systems, 1992, pp 541-547

[14] R. Bentley, T. Horstmann, J. Trevor, "*The World Wide Web as enabling technology for CSCW: The case of BSCW*", in Computer-Supported Cooperative Work: Special issue on CSCW and the Web, vol. 6, 1997. http://bscw.gmd.de/

[15] M. Robinson, "*Design for Unanticipated Use*" in: G. DeMichaelis, C. Simone, K. Schmidt (Eds.), Proceedings of the Third European Conference on Computer Supported Cooperative Work, Milan, 1993

[16] G. Hofstede, "*Culture's Consequences, International Differences in Work-Related Values*", SAGE Publications, California USA, 1982

[17] J. Pasch, "*Software-Entwicklung im Team*", Springer Verlag: Berlin Heidelberg, 1994

[18] C. Floyd, "*STEPS-Projekthandbuch."* Universität Hamburg, 1992, und 'Arbeitsunterlagen zur Lehrveranstaltung Einführung in die Softwaretechnik.' Universität Hamburg, 1994

[19] R. E. Grynter, "*Recomposition: Putting It All Back Together Again' Activities*" in Proceedings of the CSCW 1998, (Seattle, November 1998) ACM Press

[20] V. Bellotti, S. Bly, "*Walking Away from the Desktop Computer: Distributed Collaboration and Mobility in a Product Design Team*", Proceedings of the CSCW 96, ACM, 1996

[21] E. Rønby Pedersen, T. Sokoler, "*Awareness Technology: Experiments with Abstract Representation.*" In Proceedings of the HCI International '97, San Francisco, 1997. Elsevier Science Publishers, 1997

[22] J. O. Coplien, "*A Generative Development-Process Pattern Language*" In: J. O. Coplien & D. C. Schmidt (eds): Pattern Languages of Program Design. Addison-Wesley, Reading, 1995

[23] Y. Dittrich, K. Rönkkö, "*Talking Design - Co-Construction and the Use of Representations in Software Development*", Proceedings of the IRIS 22, August 1999, pp 267-279

[24] M. Rouncefield, J. A. Hughes, T. Roden, S. Viller, "*Working with 'Constant Interruption': CSCW and the Small Office*", Proceedings CSCW '94, Chapel Hill: ACM Press, 1994

PAPER IV

"Talk to Paula and Peter - They are Experienced"- The Experience Engine in a Nutshell

Conny Johansson, Patrik Hall and Michael Coquard

Accepted for publication in G. Ruhe, F. Bomarius (ed.): Learning Software Organization - Methodology and Applications. Springer-Verlag. Lecture Notes in Computer Science, Volume 1756 (appears 4th quarter 2000)

Abstract

People working in the Software Engineering field must solve problems every day to get their work done. For successful problem solving, they need to find others with specific experience. To help individuals with this, we created an 'Experience Broker' role in the organization. The broker must be officially recognized, and function as a facilitator for the internal human network in the organization. Our approach to the 'Experience Factory', what we call the 'Experience Engine', focuses on mediating referrals to sources holding the correct expertise—usually human sources.

We claim that the most valuable experience is tacit and stored on the individual level. The most valuable experience transfer comes while on the job when the person holding the experience verbally relates to the learner directly. Getting the right context for this experience transfer is difficult without direct contact between the person with the experience and the recipient. Experience is transferred at the right time when the learner is actually working on something that concerns it.

Values must be rooted in the organization, where individuals are willing to spend the necessary (most often short) time to spread valuable experiences to others in the organization. Our pilot project showed that a measurement and database approach is less valuable than individual face-to-face meetings, to transfer experience.

1. Introduction

The 'Experience Factory' [1] is an important approach for software companies in their struggle for survival. We felt it important to relate this approach to the current situation at the company, and to its organizational learning processes. We also believe that gradually integrating new ideas, concepts and approaches is the smoothest and best way for our company to apply such an extensive approach as the experience factory.

Ericsson Software Technology AB has several business centers to develop software products, called nodes, which establish product requirements and have primary customer interface as well as research and corporate functions. Each business centre is divided into several business units, each normally working on one or up to three products. The business units have 20 to 30 employees each, for a company total of approximately 800 employees. Together, they develop a wide range of software applications, including several types of telephone switches, and mobile phone management systems.

This paper starts by describing the similarities and differences between the experience engine and the experience factory. The pilot run in 1998 is presented, followed by a section describing the narrower scope and focus of our measurements. The Experience Practice describes the basic values for experience transfer, then the role of experience broker and communicator are defined, and finally we describe the different experience occasions we identified.

2. The Experience Engine vs. the Experience Factory

We believe that in applying the experience factory approach, the same applies with the software engineering discipline—these are laboratory sciences and we must experiment with techniques to see if they really work [2]. We realized rather quickly that we could not implement the entire approach, at least not directly. The name, 'Experience Factory' can lead to associations with industrial production. But software engineering is not about reproducing the same things—every product is different than all others [2]—so we renamed our application 'The

Experience Engine'. We wanted to create an engine that would continuously work to recycle and transfer the collective experience within the company.

The experience factory focuses on process improvement. And it should not be part of any project organization, since its focus does not normally coincide with that of any other part of the organization. The experience factory should be an independent part of the organization that collects, analyzes, generalizes, formalizes, packages, stores, retrieves and reuses collective experience [1]. It can provide a project organization with two kinds of experience: one that hints at what to do and the other that guides in doing it. Further, it functions to investigate and introduce new technology to the company, is the skills center for software engineering, and has operative responsibility for process improvements.

In contrast, the experience engine does not take this broad approach. That would require a larger reorganization and defining a new separate group, so established concepts and groups at our company would need to be redefined. Relying on Hansen et. al. [3], we feel it would be hard to integrate the experience engine if it were a separate functional department. The experience engine focuses on creating and managing experience transfer, integrating it with the existing organization, and the role of the individuals involved when this takes place. It does not focus on organizational structure—the roles are easily integrated with the existing organization.

The experience engine organization is, therefore, not as large as that of the experience factory. Our organization currently has one full-time and two half-time positions. The considerable difference in focus on measurements and quantification, is one reason for this. In our approach, generalization and customization are handled when the transfer occurs, face-to-face. This, as opposed to documenting the generalizations in writing, as required in the experience factory approach (as we have interpreted it). Hansen et. al. [3] declares that companies that offer customized products should not focus on measurements and store these knowledge in a database. This strongly supports the findings at our company and the characterization of our products. Furthermore, if the company has a business strategy that is based on innovative products and comprised of people who rely on tacit knowledge to solve problems, it can even quickly undermine the business, Hansen et. al. [3].

Other activities also not included involve new technology infusion, skills management, and process improvement. They certainly exist in some form, but are organizationally considered on the business unit level, and so were excluded from the study. For example, process improvement activities are normally initiated by improvement proposals reported orally, or in intermediate or final reports. Our experience is that the best way to carry out process improvement is to include it as an activity in each project. These are coordinated and followed up at the business center level by a manager, with the results communicated to the rest of the organization primarily through informal channels.

One empirical study, related to our pilot, reports that the information system at our company is rather well developed. Formal communication channels such as electronic mail and documentation (written reports) have been well established for several years. But, this might be at the expense of verbal communication. Establishing a corporate culture that includes a coordinating mechanism for communication (to transfer knowledge and experience) is essential for employees' knowledge-creating activities [4].

Taking the entire experience factory approach would have been revolutionary at our company. We also thought it would not bring any improvement. First, it would require an extensive re-organization involving definition of an experience factory organization. Second, it would require unnecessary changes to work routines that are already perceived to work well. Finally, it would involve introducing a company-wide approach affecting all activities in the organization without any easily identifiable cost benefits. And, it would probably hinder the success of our other more important efforts where we did find shortcomings and a need for improvement.

3. The Pilot Project

The pilot project we set up was limited to a few business centers, and was designed to test our ideas and evaluate their cost benefits. We continuously refined, removed, and added new guidelines and working practices as we gained experience and feedback.

This pilot empirically covered over 100 experience transfer occasions in the organization. These involved a wide range of experience transfer related to management, technical, as well as business related issues.

Management experiences were related to project management issues such as planning, progress reporting, and requirements change processing. Examples of technical experiences are design and code decisions, strategies, and solutions. Negotiations, agreements, and patent are examples of business issues addressed.

We felt it important to make a qualitative study. So instead of quickly analyzing a large number of occasions, we selected about 20% of the experience transfer occasions, and studied them in more detail. We felt the studies should be related to the context of the experience occasions. We generalized these experience occasions to determine common patterns, further developing our approach.

The project consisted of a core team with four persons working part time. All project members have a Software Engineering or Computer Science degree and several years of professional experience in software development.

The experience occasions were evaluated continuously using attitude surveys, both written questionnaires and oral interviews. During the evaluations, we identified problems with using written questionnaires. Without doing any quantitative analyses of the quality of the questionnaires—we identified several questionnaires that had been filled out in a hurry. In our view this produced unreliable results from the surveys. We have therefore started to use oral interviews, short occasions not planned in advance, where the interviewer orally goes through the questionnaire with the interviewee, who sometimes does not even see the questionnaire.

4. The Measurement Approach

Measurement is the process of assigning numbers or symbols to attributes of entities in the real world so they can be described using clearly defined rules [5]. Measurements in software engineering can be direct (as in cost or number of faults) or indirect, by combining different measurements (such as for productivity or fault density). The usefulness of a measurement depends largely on the control status of the process when it is collected [6]. This is the cause of variances observed during the process. The problem is that the causes of variance are rarely analyzed. The reasons we found for this are:

- When an organization starts measuring, its approach to measurement is too comprehensive, so too many measurements are defined,
- The necessary activities for analyzing the causes for variances are not completed,
- Feedback from analysis of the causes of variances is poor,
- Improvement activities initiated because of the measured variances are not followed.

Today, some companies still think they can put all their corporate knowledge on a server to form a kind of giant hyperlinked encyclopedia [7]. The real value of information systems is connecting people to people, so they can share their expertise and knowledge on the spot. We started with a significant effort to determine how measurements should be collected, packaged and stored. While the original purpose of the experience factory does not lend itself to creating a large database, this trap is easy to fall into. The trap is set by the descriptive illustrations of the experience factory, which often indicate storage using the now old-fashioned, disk storage device [8]—one easily visualizes physically storing experiences on a computer device. Our first approach included a large database, but after interviews, some research, and several work sessions, we concluded that this was not a good solution for the company.

In fact, we spent nearly 1000 hours on database specification before we realized this was not right for us. We determined this from the fact that over the last years we have gathered and stored large amounts of data in a large corporate database. This includes hundreds of measurements like lead time, planning vs. actual, productivity, and more. These data have rarely been analyzed for use in process improvement. Some of our time was even used to identify additional data to collect. When we presented yet another database specification, most everyone sighed at our approach. In the end, we found it more important for analysis and feedback efforts to gather a small number of different measurements. The time spent on these activities can at first be seen as wasted, but it was actually well worth finding the right direction to proceed.

We decided to narrow the scope of the measurement program. At the company level we focused on fault data and defining collection routines, storing principles, analysis methods (including statistical methods), activity definition (including responsibilities for monitoring the

activities), and simple tips on how to provide feedback on the data and fault measurement results. We focused on fault data at the company level because this is the only general measurement we saw as worth collecting, now and in future. This approach is also supported by Demarco [9]. Additional measurements and local adaptations to the corporate approach were dealt with by the local organization.

The measurements are, to a greater extent, now used more as one of several indicators of a potential problem. We have de-emphasized measurements from being the 'only' basis for decision. Other indicators can be short verbal or written analysis sections for reports, but most often, involve communication within the group.

The measurements were designed to help develop a method for variance analysis and forecasting. These primarily concerned forecasting the number of faults in operations. We chose the standard method of correlation of metrics using simple linear regression technique [5]. We identified a few interesting correlations, but we now strongly doubt these can be of any practical use. This is mainly because:

- The measurements were collected during a five year period, when products, processes and people have changed significantly,
- There were several different 'unexpected' events, such as late requirement changes, resource reductions, and poor product usage during the measurement period, which resulted in the measurements becoming unreliable,
- Uniformity of the measuring methods also arose. These included different classification systems for faults and different definitions of measurement periods.

We are convinced that different projects cannot be comparable using measures only. The projects vary too widely in size, complexity, and project member's level of experience. We found it difficult to formalize the experience not only in pure numbers, but also in writing.

We also developed a process to support the measurement cause analysis. This simple process describes how to conduct the analysis of measurement deviations reported in a project. The process includes team member interviews and evaluation sessions to determine the cause(s) of the deviation. We noticed that even though a measurement deviates from the expected, this is not necessarily unacceptable. Special condi-

tions in the project can cause deviations but these should not always be seen as indicators for action or other decision.

5. The Experience Practice

Our company values are based on a concept our senior management calls Guiding Lights. The five Guiding Lights, short phrases symbolized as lighthouses, show us the best course to follow and help us in our voyage into the future. One Guiding Light is 'We learn from each other'. This concept is emphasized to a greater extent in our organization since the ability to share knowledge and experience is most often an important aspect in salary negotiations between individual employees and their managers.

The goal of the experience engine is to transfer experiences efficiently through people building networks. To support this, we decided to define what an experience practice is. In doing so, we used the basic value contained in the concept 'We learn from each other':

The most useful experiences are stored in every individual's brain. This is the primary storing media, and verbal communication (preferably in face-to-face meetings) is by far the best way to communicate and spread these experiences.

Information and knowledge are sometimes distinguished [10]. Information expresses knowledge but is unreliable in transferring it. Knowledge should be transferred when the receiver is actually doing something related to the experience being transferred. Knowledge can be defined as: "a capacity to act" [10]. We consider experience as: "knowledge that has been perceived by acting". You can gain knowledge by reading specialist literature, but you can not gain experience without acting in practice. The main focus is to increase competency, and so transfer knowledge—a transfer mechanism which we consider equivalent to transferring experience.

This paper does not discuss in detail the difference between knowledge and experience. It is impossible to transfer knowledge precisely, but if the knowledge is merged with experience(s), it will most probably enhance competence. We use only the word 'experience' further, and do not consider the possible differences in the knowledge transfer and transferring experience.

Orr [11] states that experience is a socially distributed resource, stored and spread primarily through an oral culture. Interpreting raw measurement data is difficult without extensive routines to classify data and other context related definitions and restrictions. Written and stored information is barely recognizable to outsiders, so the author usually has to be contacted to understand the context of the experience [11]. In a face-to-face meeting with the source, the receiver can get immediate response to any questions and clarification concerning relevance, context and detail—whatever seems appropriate for the situation. In rank of importance, face-to-face meetings are the most valuable media for transferring experience—followed by telephone, video conferences, email, personally addressed mail, and finally impersonal mail such as formal reports [12].

Useful experiences are normally stored on the individual level. We can state, as Polanyi [13] and Nonaka and Takeuchi [4], that the most valuable experiences are tacit, and it is very hard to express these in words or numbers. The benefit and gain from spending time in trying to write down these experiences is hard to measure. The time it takes to transform this tacit experience into explicit experiences and write it down, is valuable and doing so often does not pay off.

Time must be taken by the author to interpret their experience, write everything down and then formulate the context. The person reading about this experience, in turn, has to spend the time to read and interpret the experience related. Several links must be performed in a uniform way to get the same result at both ends. We judge this unlikely in most cases—it is hard enough to express and transfer tacit experiences orally. But of course, we do not extend this concept into absurdity. Writing down specialist literature, documentation for standardized products or methods, and explicit (or apparently explicit) knowledge still serves a purpose.

Another important aspect is based on the axiom demonstrated by the postman game, which Swedes call the whispering game. If one person tells another an experience, who, in turn, tells it to another, and so on, the original experience related most probably differs greatly from the final version. Each individual adds, changes and subtracts their own interpretations. This is analogous to the example of the written material described. Our finding in this case is that the most valuable and useful experience transfer comes when told by those who have actually gained the experienced themselves.

If experience is not stored in writing, it will disappear when the person who actually gained it leaves the company. This is, of course, negative, but we consider that:

- Experiences which can be unambiguously interpreted are (hopefully) stored in final reports,
- The life span of the experience in software engineering is short, and
- There is a fair chance that when our experience engine approach is established, the valuable experience has been communicated to another person, and it has been merged into their own experience, so it can be shared (in its new form).

To address this last point, we made a qualitative study of the company learning process. In order to understand the organizational learning process, the entire company must be considered, not just temporary projects [14]. We studied a number of projects in a limited part of the organization, but consider the findings applicable for at least the majority of projects at our company.

Our study indicated that individuals review and observe experiences from different perspectives, and so, reflect on it to a greater extent. Further, they interpret and use their observations for problem-solving and decision-making. These are used as a base for acquiring more concrete experiences, which in turn forms the basis for more learning. These practical learning processes are similar to the cyclic learning process described by Kolb [15]. We think this indicates that experience is not as tied to specific individuals during its (longer) life span, as we initially thought. As well, people do not depend on older experiences since these are merged and refined into new, fresher understanding.

Strong management commitment is necessary to succeed with our approach. Senior and mid-level management must actively and continually support the experience engine and the principle it builds upon. Management support should not only include communicating information, but also action in continuously facilitating use of the experience engine.

6. Experience Occasions

An important question was: 'When does experience transfer take place?' Our study indicates that much experience transfer takes place during informal communication. We believe common values can be developed to support such informal experience transferring. Shrivastava [14], among others, supports this. The transfer is dependent on the work environment, time, co-workers, and the type of transfer involved. We believe that direct communication is more suited for informal experience transfer than text.

What happens when two, or more, people meet and talk depends on their interpretations, language, personalities, and the context. The basic values for communicating experiences already exists in the organization and instances of experience transfer occur rather frequently. When introducing the experience engine, we saw that experience occasions could be identified in three categories, which we called 'Flashes', 'Learning Situations' and 'Education/Training' to more easily differentiate them. Flashes and Learning Situations are described in this section. Education/Training is when the learner lacks formal training and/or professional experience in a specific area. The experience engine is not designed for education/training situations.

Of course, whether an experience transfer belongs in one of these categories or the other is always debatable. We created these groups simply as practical guides to clarify them, but they are definitely not intended to formalize any relationships or be used as rules. In our pilot study we estimated there were 20 learning situations and 80 flashes in 1998. These are only estimates since we could not register some flashes and learning situations before being fully aware they had occurred.

6.1 Flashes

Flashes are the most common experience occasion. We identified two types of flashes: 'Broker Flashes', where the referral to an experience source is mediated; and 'Communicator Flashes', where experience is actually communicated. Most flashes were internal to the organization, involving employees. However, we see no limitation that others outside the organization can be involved in this type of experience occasion. We

see this as a matter of mutual needs where there is no direct competition between the organizations. If we help you this time, you help us next.

Experiences are transferred most effectively when the receiver is performing in the process concerning the experience actually required [10]. Experiences are valuable only when they can be contextualized, decontextualized, and recontextualized at the proper time [16]. Broker Flashes do happen through written messages (mostly email), but occur more often in telephone or face-to-face meetings.

Face-to-face meetings are short occasions that usually happen by chance resulting from actual needs. In listening to people as they discussed everyday problems in corridors, on breaks, and at lunch, brokers can act as agents for those needing extra, experienced advice. It is at these short (usually only a few minutes), recurring occasions where we acted as middleman in the human network. However, to reduce relying on chance, we could create broker flashes by actively seeking them. These flashes could be created by the broker:

- Walking around - dedicating time to just walking around specifically to talk and listen to others,
- Participating in reviews and inspections – these are designed for concrete active help to the receiver so this activity is therefore easy to adapt,
- Participating in project risk analyses,
- Participating in different kinds of cross-section teams and networks.

We defined communicator flashes as taking less than 10 minutes. They have sometimes occurred previously but only in limited sized groups, such as teams or units of less than 30. The conversion of the tacit experience to explicit experience—that is, interpreting and the opportunity to use the experience which is transferred, is occurs at the same time as the transfer. Explicit concepts can, for example, be created through dialogue using metaphors and analogies [4]. Communicator flashes do not cost the receiver anything. Examples are questions and problems regarding specific programming language constructs, product storage handling, and test case specifications.

6.2 Learning Situations

Learning situations are when the receiver has formal training but lacks professional experience. This situation normally requires planned time and takes more than 10 minutes. Learning situations involve on-the-job training where the receiver learns from doing in actual conditions, but in a session with an experienced communicator. This is also known as experience transfer by tradition. And is reported as being up to seven times more effective in transferring experience than the most common method—the lecture [10]. The guidelines used for the conversion of tacit experiences to explicit experiences in communicator flashes also apply to learning situations.

Learning Situations can be when the communicator:

- Moderates a work-shop,
- Participates in developing a project plan with the receiver (for the receiver's project),
- Guides a quality coordinator in using company software quality assurance processes.

The best approach we found for calculating cost benefit was using simple attitude surveys on a case-by-case basis. These attitude surveys were done with either interviews or questionnaires. One example of a learning situation is where the communicator participated in a three hour walk-through of a preliminary study. The receiver reported the walk-through (involving only the communicator and the receiver) as successful, estimating a savings of several hours and a considerable quality improvement gain.

This highlights our feeling that we are not only interested in actual cost benefits, but also other organizational and individual benefits. This is often difficult to measure in numbers so we felt the approach of estimating benefit using these attitude surveys could provide evidence of success. However, we do make a careful attempt to transform this 'tacit' understanding into explicit numbers. Each learning situation and communicator flash is, where possible, graded by value in a 10 point ordinal scale (1 to 10), and by estimated cost benefit (estimated number of hours earned).

Regarding another aspect, when companies pay for a service, they have to evaluate the results of the work performed. They should be keen

to optimize the use of the time paid for, whether they buy it internally or externally. This argues strongly for the receiver to want to pay for the learning situations. Also, from the perspective of the communicator (and the financial unit they belong to), it is important that the learning situation does not affect their budget negatively. Follow-up for both costs and benefit would therefore be simplified if these situations are budgeted.

Again, we do not see any limitation for involving external personnel in learning situations. With the learning situation financed by the receiver, this is easier done than for flashes. However, the problem of allocating the necessary time for experienced communicators remains, whether this is acquired internally or externally to the learning situation. This kind of resource is scarce, especially in software engineering, so these busy people have difficulty allocating the necessary time.

7. Experience Broker and Experience Communicator

People in organizations must solve problems to get their work done every day. For this, they must often find expertise with this experience. We use the term expertise as the embodiment of experiences within individuals [17]. Individuals have different levels of expertise in different topics and in different contexts. To solve the problem of identifying the expertise required and selecting who to acquire it from, two new roles were created in the organization, that of 'Experience Broker' and 'Experience Communicator'. The experience broker has a visible, appointed role within the organization. This role is similar to the expertise concierge described by McDonald and Ackerman [17] and the contact broker role described by Paepcke [18]. Brokers must be visible and accessible to facilitate helping individuals looking for expertise, when they need an experience transfer. Experience communicators, on the other hand, have a more informal role in the organization.

Those lacking the necessary experience just contact the experience broker, who helps them find the right expertise, or experience communicators, with the necessary experience. Justification of the possibility to use the transferred experiences must be done at, or closely after, the time of transfer. If the conversion does not add any value or is not

enough to be able to generalize and adapt it to one's own context, you need to seek other or more expertise.

The problem with expertise selection is well studied and described by McDonald and Ackerman [17]. People are often faced with choosing from among several possible experts. Additionally, if they have a complex problem they will most likely seek more expertise in order to achieve a higher probability of quality in selecting experience communicators, and/or generalization of the experiences [17]. There will be many experience communicators but only a handful of specifically appointed experience brokers, connecting them all in the experience engine.

7.1 The Experience Broker

Experience brokers add some extra power to the engine to make it run better. They function as the hub of the experience engine network. Their primary responsibility is to maintain, extend, and facilitate using the experience engine network and its experience communicators. These are generalist with wider experience in using tools and methods–both within and outside the company. So they know a little about many different things, but can easily understand the company structure (products) and its processes.

In this sense, the experience broker is teaming—someone who listens, sometimes even eavesdropping, but who also talks with many people, referring those who need it to the right experience communicator(s). This is not to say they spy on employees, but they invest their time trying to find out what others' needs really are. This can be thought of as taking the organization's temperature, finding out what is going on, and solving networking problems. The broker therefore knows the projects currently running, even participating in formal activities for selected projects (including meetings) to extract what experience is needed.

The broker knows the arenas where people interact and should also try to find or create additional arenas for transferring experience [4]. This role is essential in creating a company learning climate.

The experience broker's role is to:

- Identify several experience communicators who can provide others with help/experience,

- Facilitate contacts with experience communicators,
- Strive to connect people across organizational boundaries,
- Be perceived as always available,
- Follow up on the benefits from flashes and learning situations.

This role must be fully recognized in the company for it to succeed. This means that it is both well known, and appreciated. The person acting as a broker must have a large contact network, and be respected as a generalist in the company. This can be achieved by communicating the concept as well as success stories through both informal and formal communication, including departmental and project presentations. An important consideration for such presentations is to have advance information about the current status of the projects or units the (potential) receivers represent. This way, the broker can find concrete applications for flashes and learning situations.

The restaurant is a useful arena for mediating referrals to experience sources. In the extreme case, the broker should always try to have lunch with different people, take breaks at different locations, and participate in activities with employees outside work. In these situations the broker always works when others have breaks, lunch or spare time. This may not be fully attainable from the broker's social perspective, but these arenas for mediation and experience transfer present good opportunities.

The experience broker must have good communication and interactive skills with others. Some people are naturally this way, but these skills can also be learned. Experience brokers must have a good sense of when to interact with a project or a person, and when not to. They should not, for example, disturb a person who is deeply occupied, and they have to know where to find the right person with the necessary experience to then mediate the contact. And even more important, project members and others must see their needs met when using the experience broker.

The length of time for anyone to be experience broker was also discussed. Though we have not come to any final determination, we believe this depends largely on the individual involved. While some should stay brokers for shorter periods, others have the qualities necessary to hold this role longer. Some are suited to do this full time, while others are more suitable as part-time brokers. One benefit with part-

time brokers is that they can maintain their role as a generalist by participating in different projects with their remaining time. It is, however, of utmost importance that the organization has full confidence and trust in the individuals acting as brokers.

There are too many specialists and too few generalists at the company today. To ensure that the experience broker stays a generalist, they must be known to the various parts of the organization and different projects. One way to achieve this, is known as walk around management, which has been successfully used by great companies around the world. We feel this principle can be used for experience brokers, though they are not managers. This should be an independent position that is not assigned to any one branch or specialty.

As time passes, an increasing number of experience occasions will occur without any broker having knowledge of them. There are already several examples of mediations where one communicator, shortly after an experience transfer, is asked for additional, or related, experience. In this case, an experience transfer occurs that no brokers know of—the contact is most probably taken directly with the communicator. This can result in a problem with following up experience occasions, but can also show progress towards the goal of the experience engine—getting people to build human networks.

7.2 The Experience Communicator

Experience communicators are those with the experience who are willing to share it with others. They should have some pedagogical skills in addition to their willingness. We do not think everyone with vast experience can act as a communicator, rather it is equally important, as with brokers, to have human understanding and social skills.

We found it important for experience communicators to try to give educational answers. These answers are focused on teaching receivers how to solve related issues by themselves in future. Lao Tzu is credited with coining a useful proverb: "Give the man a fish and you feed him for a day, teach the man how to fish and you will feed him for a life time". Providing educational answers means placing the answer in a broader context, or even just hinting at where to find the answer. For example, an answer may be easily available on the web or in a report. Otherwise, providing receivers with clues on how to solve the question

themselves may be the best approach without further describing the detailed context.

Experience communicators should not do others' work. Rather, the communicator's main task is to help others solve their own problems. This is different than the role of a regular project resource. While using the communicator as a resource in a project may help the project in the short-term, it would damage the entire idea behind the experience engine. Little or no experience transferring would take place in such situations.

The role of experience communicator was an informal role in the organization. And we are strongly convinced that the role should remain so. The company already has a lot of positions and roles, and attitude surveys show low acceptance to introducing yet another role. However, we are considering how this informal role should be made more visible in the organization. The benefit to the communicators would be that they are perceived as valuable, attaining a degree of recognition in the company. We believe that an important factor would be to high-light the company value behind 'we learn from each other', and include it as an issue in salary negotiations.

It is also important for the experience communicator to give feedback to the experience broker. This not only ensures that processes can be reviewed, but also that the communicator actively takes part in the experience engine. It is equally important for the broker to know which communicators add value to the experience engine. For example, feedback from the communicator to the broker could be: "I have had many people calling me about the same type of problem. I think we need some courses in this field," or "I was asked a difficult problem that I could not resolve. I referred them to another person I know who is good in that field." This fits the goal of the experience engine: to get people to build human networks.

It is important to realize that everyone is not suited to be an experience communicator. Everyone does not have the aptitude and the social skills necessary to transfer their experience pedagogically. On rare occasions, we have had problems with evaluating an individual's suitability to be communicator, but we believe this can be handled in normal procedures within the organization.

8. For the Future

We will continue to refine our successful approach, focusing on mediating referrals to others as a source of collective experience. A five-fold increase in the number of mediations during a three month period gives an indication. We plan to consider integrating more of the key features described in the 'Experience Factory' approach into our experience engine. In particular, process improvements will be a key issue for this. The extension of our approach will also most probably mean the organization will grow by one or two persons in the near future.

The five-phase model of organizational knowledge-creation process described by Nonaka and Takeuchi [4] will be studied. So far, we have only studied justification of experiences at the individual level. However, the need for an organization to conduct a justification on the organizational level [4] will be considered, to perhaps be introduced for some experience occasions.

Currently, we are incrementally extending use of the experience engine in the organization. Some spontaneous mediation and experience transfer takes place throughout the company, of course. But we plan to introduce it step-by-step. Experience engine implementation issues are currently being discussed—for example, is it necessary to formally define the role of the experience broker and will we still be able to reach the goals of the experience engine? We plan to allocate more resources. The (informal) list of communicators has weekly additions. We also need to define and resolve the issue of recognizing the experience communicators.

As well, we need more success stories focusing on cost or lead time improvements. Success stories are one of the most important incentives to apply any approach. For our part, we want more stories that emphasize using project managers and team members, as well as success stories useful for senior management.

We found that the best solution to calculate cost benefit was to determine the benefit through attitude surveys, while not concentrating on cost only. We will continue to evaluate the communicator flashes and learning situations for perceived value, and estimated cost benefit using estimated number of hours earned. We strongly believe that these evaluations should be used with great care. They can be used as indicators but most probably should only rarely be used alone in making decisions on direction. These measurements are, however, necessary to

provide indicators of benefit for the company, especially to senior management, from using our approach.

9. Conclusions

Using our 'Experience Engine' approach, we aim to eliminate the attitude that managing collective company experience involves huge databases and processing large amounts of measurements. Our approach favors verbally communicating valuable experience and de-emphasizes the practice of collecting too much data. This relies heavily on values of willingness to share experiences and to mediate referrals to others who hold the necessary experiences. It also relies on two important roles, experience brokers to mediate the referrals, the 'broker flashes', and experience communicators to transfer the actual experience. We use the terms 'communicator flashes', 'learning situations' and 'education/training' for experience occasions. The experience engine handles communicator flashes—short experience transfer occasions which take only a few minutes, and learning situations—longer sessions with experienced people, when the receiver has formal training but lacks professional experience.

The experience broker is a generalist who helps others in finding the right expertise with valuable experience. The broker is visible within the organization, and has a wide, general knowledge of methods and tools. The broker is active in arenas for mediating referrals to sources holding experience, and also tries to find and create new similar arenas.

The main task of the experience communicator is to help other people to solve problems. The communicator focuses on giving pedagogical (educational) answers to teach the receiver how to solve related issues on their own. The broker and the communicator must have an aptitude for interacting, and knowing how and when to interact, with others.

One drawback with our approach is its heavy reliance on senior, highly experienced and respected individuals, to act as brokers and communicators. These individuals are the most attractive ones on the market today increasing the difficulty in keeping them as brokers or communicators—or even keeping them in the organization at all. But, knowing that old experience is not as valuable as new, and that previous experiences are, to some extent, retained and combined with new expe-

riences, we feel that reasonable staff turnover will not have a significant negative affect.

We have a vision that a formal experience broker role will not be needed in an optimal learning organization. When the practises outlined in this paper mature and reach higher company levels, many individuals will be able to act as brokers without being assigned the role of experience broker. Everyone can adopt the company values, and obtain the skills needed to act as communicators. Experience occasions will occur often, simply when people meet in the corridors, at the copier, or in the lunch queue. So before they buy their meal, they will have been served either an experience they need or the name of a human source with that experience.

References

[1] V. R. Basili, G. Caldiera, H. D. Rombach, "*Experience Factory*", Encyclopedia of Software Engineering Vol. 2 Editor: J. Marciniak, John Wiley and Sons Inc., 1994, pp 528-532

[2] V. R. Basili, "*The Role of Experimentation in Software Engineering: Past, Current, and Future*", Proceedings of ICSE '96, 1996, pp 442-449

[3] M. T. Hansen, N. Nohria, T. Tierney, "*What's Your Strategy For Managing Knowledge?*", Harvard Business Review March-April 1999, pp 106-116

[4] I. Nonaka, H. Takeuchi, "*The Knowledge-Creating Company*", Oxford University Press, 1995

[5] N. E. Fenton, S. L. Pfleeger, "*Software Metrics, A Rigorous & Practical Approach Second Edition*", International Thomson Computer Press, 1997

[6] A. Burr, M. Owen, "*Statistical Methods for Software Quality*", International Thomson Computer Press, 1996

[7] T. A. Stewart, "*Intellectual Capital: The New Wealth of Organizations*", Brealey Publishing, 1998

[8] PERFECT Consortium, "*The PERFECT Experience Factory Model*", PERFECT Esprit-III Project 9090, 1996

[9] T. Demarco, "*Why does Software Cost so Much? And Other Puzzles of the Information Age*", Dorset House, 1995

[10] K. E. Sveiby, "*The New Organizational Wealth*", Berrett-Koehler Publishers Inc., 1997

[11] J. E. Orr, "*Talking about Machines - an Ethnography of a Modern Job*", IRL Cornell Press, 1996

[12] R. L. Daft, G. P. Huber, "*How Organisations Learn: A Communication Framework*", Research in the Sociology of Organizations Vol 5, pp 1-36

[13] M. Polanyi, "*The Tacit Dimension*", Routledge and Kegan Paul, 1966

[14] P. Shrivastava, "*A Typology of Organizational Learning Systems*", Journal of Management Studies Vol 20 No 1, 1983, pp 7-28

[15] D. A. Kolb, J. Osland, I. M. Rubin, "*Organizational Behavior: An Experiential Approach*", Prentice Hall, 1995

[16] M. S. Ackerman, C. Halverson, "*Considering an Organization's Memory*", Proceedings CSCW 98, 1998, pp 39-48

[17] D. W. McDonald, M. S. Ackerman, "*Just Talk to Me: A Field Study of Expertise Location*", Proceedings CSCW 98, 1998, pp 315-324

[18] A. Paepcke, "*Information Needs in Technical Work Settings and Their Implications for the Design of Computer Tools*", CSCW 1996, pp 63-92

PAPER V

Surprising Results from a Measurement Study - is Software Measurement an Exaggerated and Over-emphasised Area?

Conny Johansson

Published on CD-ROM in the 3rd European Software Measurement Conference FESMA-AEMES 2000, October 2000, Madrid, Spain

Abstract

Many software companies are involved in eager attempts to manage experience and knowledge on an organisational level. Experience that can be used for estimating and planning purposes are of great value. One obvious solution can be defining and storing a set of measurements representing these experiences, carry out statistical analyses, and developing rules and guidelines to find and use this explicit knowledge. This four-year study using student projects indicates that the great majority of these measurements, seemingly unambiguous, include several uncertainties. We found it difficult to generalise the measurements since many factors have a different influence on each. The experience database we designed contains only three planning constants. The part of the study adding significant value to process improvement efforts was the analysis of software faults found in testing. Software faults were analysed for description, consequence, and proposals for actions to avoid similar faults reoccurring in the future. The potential for improvement resulting from the actions proposed is deemed high, and can probably reduce the number of major faults later on. Statistical analyses did not, however, prove useful because of all the uncertainty factors. Detailed analysis, primarily performed in seminars and face-to-face discussions, was necessary to interpret and understand the measures and statistical analyses. We are sceptical about using statistics due to all

the difficulties associated with interpreting each measure, and since it is too tricky to perform a complete and fully satisfactory analysis of all existing uncertainty factors.

1. Introduction

Managing knowledge-identifying, analysing, storing, sharing, and transferring it-is important to the corporate learning process. Accomplishing this has been identified as important to succeed with business strategy [1]. Knowledge has become the resource, not a resource, and researchers have identified an increased focus on knowledge as a competitive resource [2]. Even when companies are superior to their competitors technologically, they often find it difficult to handle knowledge and experiences within the company [3]. The Experience Factory approach [4], a well-known knowledge strategy among software companies, emphasises the codification approach [5] as described by Hansen et al [6], and is characterised by knowledge management centred on the computer. Knowledge is codified and stored in databases and used by anyone in the company. In contrast to codification, others describe the personalisation approach [6], which is characterised by knowledge closely associated with the individual who developed it and is primarily shared person-to-person. The chief purpose of computers is to help people communicate knowledge, not to store it.

The limited scope of this study did not consider any possible differences in preserving and communicating knowledge and experiences. Taking a look at the glossary, experience is defined as knowledge that results from direct participation in events or activities, while knowledge can be gained without active participation. We do, however, consider the difference between explicit and tacit knowledge. Explicit knowledge can be expressed by words and numbers, while tacit knowledge is not easily expressed and is thus hard to formalise [2]. This study was focused on what we assumed were explicit experiences, and we adopted the codification approach from the very beginning.

We hypothesised that it was possible to formalise general experiences and to draw conclusions from statistical calculations. Relying heavily on the Capability Maturity Model's (CMM) [7] emphasis on quantitative management, the ultimate aim was to design a measurement database of real experience from projects. This experience could then be used for

estimating and planning purposes and perhaps for quality evaluation of the product developed, also.

2. Background

This study comprised 38 student projects, which were executed during a four-year period from 1996 to 1999. The projects are a part of an undergraduate course in small team software development. The emphasis regarding software development in this course is on requirements quantification, estimating and project planning, progress reporting and system testing. Each project consisted of four or five persons, had a lead-time between 16 and 18 weeks, and a total budget of 280 hours for each individual.

The participants in the project have experience corresponding to one and a half years of study in Software Engineering and Computer Science at undergraduate level. Examples of courses the students have attended include programming, analysis and design, discrete mathematics, database technology, real time systems, and operating systems.

A project course in individual software development, which provides the basis for professional software development, must have been passed. Negotiations with clients and simple contract processing are practised in a real environment. Representatives from companies acted clients, usually for new products in an innovative area. The codification part of the individual project course is restricted to a few quantitative measures (such as fault and cost data), and the personalisation part of a final oral presentation of problems and difficulties experienced during the project. The small team projects each have a successor in a large team project that the students attend 6 months later [8].

Characterising the projects included in this study gave the following:

- 70% of the projects used PC and Windows for the development and target environment, 26% uses Workstations and Unix, and 4% other environments, (approximately one third of the projects were in fact independent of environment),
- 60% of the projects developed interactive systems, and where the main part comprised communication solutions, 21% real time systems, and 19% aimed at developing platforms and tools,

- 47% were developed in C++, 32% in Java, 10% were developed only using database tools, and 11% using other kinds of development tools such as Visual Basic, (90% of all projects included database solutions).

The products the projects develop are of various kinds. Examples are mailing list services using SMS, a handheld based system for registration of articles for electricians, and a framework and platform for server based multi-player games.

3. Ensuring Uniformity

3.1 General

The usefulness of measurements depends largely on the control status of the process when they were collected [9]. Uniformity between the projects regarding the application of processes used, and the knowledge and skills of the team members are important factors. It is easier to achieve uniformity among students than among professionals, since they have a more common background and processes can be enforced more easily in non-commercial projects. Special effort was made to establish a working culture, to define the roles to be occupied, and in designing the software development process-all to reinforce uniformity. The budget for the measurement activities, which was approximately equal to the actual time spent, corresponds to approximately 8% of the total cost for a total of 4000 hours for all 38 projects.

3.2 The Working culture

Establishing a corporate culture that includes a co-ordinating mechanism to communicate (that is to transfer knowledge and experience) is essential for employees' knowledge-creating activities [2]. Humphrey's commitment ethic [10] is a good base to orient individuals towards an enterprising attitude that is mature and aware of their responsibility. The commitment culture we adopted is described by Ohlsson and Johansson [8], but the issues we emphasise are:

- Agreement on what is to be done,

- Team members have the chance to influence requirements through negotiations,
- Individuals try to meet the commitment, even if help is needed,
- If it is clear that this cannot be done prior to the committed date, then advance notice is given and a new commitment is negotiated.

3.3 The Roles

The course is designed to simulate how a real company works, or should work, in order to succeed with their projects. This includes the definition of roles within the project. The approach is described in detail in [8]. Internal project roles, except the project manager role, are left to the students themselves to decide on. On the global level, we have defined three different roles not allocated to students: the department head, the client, and the design mentor.

The Department Head. The department head, a teacher who is supervising the project, is employed at the same 'company', the supplier, as the team members. She/he ensures that the internal requirements on the processes are followed, and supports the project when problems arise. Any difficulties such as team and client problems, activity planning, and risks and problems, should be openly and honestly discussed with the department head. Meetings were held every week where weekly progress reports are discussed.

The Client. The client is a representative from a real company who negotiates the requirements. Important goals for the project are to keep the final delivery date, fulfil the requirements agreed on, and have no (unacceptable) faults in the delivered product. The most important client quality is for them to act naturally-as real clients.

The Design Mentor. We are convinced that the design, primarily the overall system architecture, is the most important technological part of a maintainable product. The details of the technological foundation-object-orientation and the learning process-are described in [11]. We created a specialist role filled by a teacher who could be consulted for 10 hours during the project. The design mentor's responsibility and authority are defined, such as she/he shall approve the overall system architecture.

3.4 The Software Development Process

We have created a fairly simple development method focused on a few rather traditional phases: requirements analysis, estimating and project planning, system test planning, system design, implementation, system test, and acceptance. The method is result oriented where each phase contains a milestone to achieve, defining the results to achieve in each phase. Mandatory results include the requirements specification, a contract, a project plan, estimation documents, progress reports, a system test specification, system design, documented code, a system test report, and a final report. The criteria for each milestone vary, but generally described are:

- the client has formally signed an approval of the result,
- a result has been reviewed and accepted by some or several of the project participants,
- a result has been reviewed and approved by the department head.

4. Fault Analysis

4.1 The Analysis Approach

Software faults are static program characteristics that cause software failures [12]. Faults are not only programming defects, but also unexpected behaviour where the software conforms to its requirements though these are incomplete in themselves. The formal verification activity we used was the system test, defined as the activity where the product is tested against the specified, quantified, goals stated in the requirements specification. Categories for system testing include functionality, use cases, capacity, response times, security, and performance.

In order to learn something from the existing software faults, you must be aware of what faults you have made. Measuring faults and analysing their causes provides input to create a good base for planning future fault prevention strategies and activities. The activity of analysing fault causes focuses mainly on faults found in system test, but can also include major faults found in any other test activities in the project. The analysis had four main activities:

- Why did the fault occur (what are the sources of the fault)?
- What were/would the consequences have been?
- What should be done to prevent the fault from reoccurring?
- How much did it cost to find and correct the fault?

Suggestions regarding solutions, improvement to processes, checklists and improved verification routines that could be used to avoid the fault from reoccurring are presented. The result from the analysis of all projects in the current year is compiled and documented and seminars are held in order to clarify descriptions, causes, and reasons for the improvements as based on the findings. This paper discusses the evaluation of all projects performed during 4 years.

4.2 Description of faults

The measurement method for the description of faults uses categorisation of fault types. Categorisation is based on the ideas of Jones [13], and each root cause is determined by studying the final report and complementary discussions to clarify the context. We have also looked at IEEE Standards [14] regarding fault classification, but we decided early on that this classification uses too many categories. Often (in fact much too often) it was difficult to do this even with a few categories, especially when the cause of the fault appears to have several concurrent sources. However, of 190 faults the sources were categorised as follows (with percent occurrence):

- implementation - 23%,
- system design - 21%,
- requirements analysis - 18%,
- project planning and estimations - 11%,
- test planning - 5%,
- configuration management - 3%,
- delivery and acceptance - 2%, and
- others (which are usually difficult to categorise) - 17%.

Compared to Jones's result, [13], the percentage of errors with source cause "requirements analysis" is high, and the value for the source "implementation" is low. The value for "implementation" is also low if you compare with Boehm's result [15]. The categorisation leads, however, to the conclusion that our thoughts about areas for improvement were verified. The measures alone are hard to interpret without any detailed analysis, though, and start from when decisions of actions for improvements should be taken. More detailed analysis of the part of the process (sub-process) were the fault occurred is necessary to be able to define more specific, useful actions for improvements.

4.3 Consequence analysis

Fault consequences were analysed and categorised according to: (a) incorrect or missing functionality, (b) unfulfilled non-functional requirement, (c) hanging software, and (d) no effect. More than 90% of the consequences were categorised in the category of "incorrect or missing functionality". Further, analysing general patterns regarding the cost of the fault includes two sub-activities. We adopted the simple analysis technique of the box plot, which uses the median and the quartiles to define the central location and the value spread. Values outside the upper and lower tails, quartiles, are called outliers. Fenton and Pfleeger [16] describe the box plot technique in more detail but it is a very useful visual form for analysing and presenting statistics. Our strategy concerning whether the outliers are removed, and whether the analysis is continued without the outliers, is not limited to determining whether the value is in the upper or lower quartile. Equally important is if the outliers seem to be unrelated with the other values. So these decisions varied from one measure to another. The two sub-activities regarding cost analysis and the result from them were as described below.

- "How long did it take to find the cause of the fault?"
  The median value was 3 hours. There are a few outliers differing widely, indicating a few faults whose cause was hard to find. We could not draw any general conclusions regarding what categories generate the faults that were most costly to find. The most significant improvement for four of the 'most' expensive faults was, however, connected to third party products. So there were too few cases from which to draw general conclusions.

- "How long did it take to correct the fault?"
  The median value was 2 hours. Three of the most expensive faults to correct were performance faults. Once more, this is too few to be able to draw general conclusions.

4.4 Proposals of actions to prevent faults

Our proposals for how to avoid similar faults in the future are of course related to the analyses of their causes.

1. *Improve communication*. Errors in setting requirements have been reported as a common problem by several authors. Further, errors in this category tend to be the most troublesome and expensive to correct afterwards [12] [13]. Despite having a signed agreement, we need to maintain good communication with the client through the entire project. Frequent contact is necessary to elicit the client's implicit requirements, which are often hard to formulate so many times these are not even formulated at all. This communication includes demonstrating prototypes and showing documents included in the final delivery early on.
   Another important issue is internal communication between the team members. This was, in fact, perceived as being equally important as communication with the client. Results should be reviewed and discussed carefully, and notes should be written so that each internal agreement that has been documented. The importance of communication within the working group is also confirmed by Humphrey [10] [17].

2. *Avoid "copy and paste" problems*. "Copy and paste" problems occur when parts of a text or a code segment are duplicated from one place to another. This may seem surprising since team member's technological base is in object-orientation, which is used partly because it is designed to minimise duplication of similar code segments. In our case, error-handling situations contributed with the majority of "copy and paste" problems. When duplicating text, we found it hard to balance between having one type of information in only one place and having the same text in several documents (as is necessary sometimes to increase clarity and overall

understanding of separate documents). Care is recommended when copying segments, including recording each time text is copied. A simple "copy record" table can help especially to avoid:

- the fault being duplicated, and
- the necessity to correct several copies during when correcting faults.

3. *Be careful with third party products*. Almost all projects used third party products when developing their product, primarily development environments and tools. These products often either did not function as expected, did not follow the specifications, or the wrong version was installed at the target environment. Similar problems are discussed by Sommerville [12] among others. We recommend securing the configuration of the target environment, and to actively verify written and verbal specifications and promises at an early stage of the development.

4. *Pay special attention to design interfaces*. Complex design interfaces are hard to manage. As stated above, we have shown the source of the root cause for system design is 21 percent. Improvements to design interfaces are estimated to have a high potential to avoid faults with other sources of faults too, such as implementation and test planning. This includes both interfaces between parts that have been developed by different individuals, and those that developed by the same person. Database interfaces, interfaces between classes, and communication interfaces between classes and methods, are often so complex that detailed knowledge of each part is necessary in order to interpret and understand them.

5. *Other improvements*. The root cause for faults whose main source is categorised under "implementation" can be roughly divided into logical and error handling faults. Tips on avoiding these kinds of faults include identifying complex code (long algorithms and logical expressions) and to divide these into smaller parts in order to achieve simplicity. Code reviews are one method of finding such complex constructions, and could complement testing.

Several errors seem to occur due to stress originating from underestimating the time needed to conclude an activity. There are indications that activities are concluded when time estimated has passed or exceeded, leading to neglected reviews and tests so that faults remain in the system.

Even though project members all were active undergraduate students, we strongly believe based on our own professional experience, that lack of competence is not a problem unique for active students-it also appears among professionals. Software Engineering is a young field and is changing rapidly so new methods and techniques always being developed. Anyone not working with new tools and technology or learning about these is quickly using outdated knowledge. Higher competency levels and greater experience are seen as having great potential for improving on the issues discussed in this section.

5. Measurements and Statistical Analysis

5.1 Preparations

Several measurements had to be collected during the projects. This was to provide information on the progress of each project and obtain knowledge and experience that can be generalised. The goal was to build a database with generalised planning constants for estimating and planning future projects. We have found it important to integrate the specific activities of measurement into the development projects, and to pinpoint the measurements to be used in projects later on (in fact they will use it in the project which starts 6 months after the current project is concluded). This approach is supported by the "Practical Software Measurement" approach [18], which is useful for selecting and applying software measures that support the project needs. Several measurements are based on comparing planned vs. actual outcome regarding lead-time, cost (man-hours) and size. Productivity measures collected include the number of lines per hour, number of hours per class, and number of pages per hour. Other measures valuable for statistical calculations are number of hours for each phase, number of requirements, number of faults in system test, and number of hours for correcting a fault.

5.2 Results from Measurements - Planning Constants

We agree with Boehm [15] of the necessity to use several cost estimation techniques, and that the technique's strengths and weaknesses are complementary. Those who did the estimations had limited experience from software projects. But by using expertise (the department head), estimation by analogy [19] and discussions with team-mates, they had a fair chance of making good estimates. We have, during four years, tried to build a database with valuable, general planning constants. But, surprisingly and disappointingly, this database really ended up in a small one. Using the box plot technique, as described earlier, we have in fact only identified three planning constants that we can rely on. These are generally described below.

- The cost for the pre-study is between 15 and 20 percent of the total cost of the project. The cost is the hours registered for the activities from the very beginning of the project to when the requirements, estimations, and planning are done, and a contract is signed.
- Productivity in the implementation phase is between 20 and 30 lines of code per implementation hour. The implementation hours are those hours that are registered on the activities classified as implementation activities.
- The time it takes to write one page of documentation is approximately two hours. This concerns activities where all kind of documents, excluding code, are produced as output. Noticeably, the effort seems to increase when the number of pages in a document is low, about one to three pages.

At first glance these few findings could be seen as a failure of the whole approach of gathering planning constants, but later on we saw it from the positive side. We have at least gained a few planning constants that we can rely on. Furthermore, we were very surprised when we could not identify any significant difference regarding the different categories. The development environment, platform, type of system, and choice of programming language (as in C++ and Java) had no differences in their measures. When a measure was applicable for several types of systems, there was no visual or statistical deviation between the categories. This could indicate that we have chosen the wrong categories or

perhaps the wrong measures so the categorisation should perhaps be changed.

Not only to collecting was all measurements time consuming but also doing the analyses and interpretation-or as Demarco [20] expresses it: "And it really cost a ton of money". We believe that the cost was much too high in relation to what we really gained. First, you must analyse if the working culture, software development process, and application of roles follow an acceptable uniformity. We have found this uniformity hard to assess and hard to achieve, even though we have carefully supervised the projects. Secondly, you must pay attention to all the factors that may impact on a measurement. The factors we have found important to analyse are requirement changes, late hardware deliveries, poorly defined milestones, changes concerning technical design, and inexperience in applying measurements.

5.3 Statistical Analysis

Measurements were designed to do statistical analysis with the purpose of developing general guidelines for estimating and planning. We inspected scatter plots and used the standard method of correlation of measurements using the simple linear regression technique [16]. This simple correlation technique takes a mean squared fit of the data and calculates the differences between the actual data and the calculated line. If these residuals are within an area predicted from a normal distribution, then the correlation has a reasonable probability of being valid. Furthermore, tests were done to investigate practical significance and to calculate the likelihood of finding the correlation by chance. These procedures, including the formula for calculating the correlation coefficient, techniques for testing the normal distribution of data sets, and the significance of a calculation are described by Fenton [16] and Humphrey [21], among others.

When deciding if two data sets have a relationship or not, we relied on Humphreys [21] rules for the correlation procedure. These rules interpret the correlation coefficient, r (-1<=r<=1), according to:

- If the square of the correlation coefficient, r^2, >=0.9 then the correlation is considered predictive and you can use it with high confidence.

- If $0.9>r^2>=0.7$, then there is a strong correlation, and the relationship is adequate for planning purposes.
- If $0.7>r^2>=0.5$, then there is an adequate correlation, and the relationship could be used for planning purposes, but with caution.
- If $0.5>r^2$, then there is no relationship, and the relationship is not reliable for planning purposes.

5.4 Result from the statistical analysis

Expectations for the statistical analysis were very high. We have used the measures described above and calculated the correlation coefficient for every relevant pair of values. However, the result was very disappointing. We identified no correlation that could be considered predictive, according to the criteria described. Two correlations could be considered strong and significant. These were:

- the number of faults in system testing increases when the number of lines of code increases,
- the number of lines of code increases when the implementation time increases.

The correlations that were strong and adequate are obvious in our view, and so added no value to the experience database. We identified one correlation that could be considered as adequate—the number of faults in system test increases if implementation time decreases. This seems to be a hint to increase the implementation time in order to avoid faults from being introduced. But we could not identify any common pattern in the detailed analysis except the statistical calculation that could verify the correctness of this correlation. Possibly the faults had too widely varying characteristics, the problems concerning requirements handling varied significantly, or uncertainties regarding uniformity of detailed time keeping were considered too great.

We identified several correlation's where $0.5>r^2>=0.4$, which if they were reliable, would have been useful when planning future projects. One example is "the time for correcting faults decreases when the time for the pre-study increases". It is tempting to interpret an inadequate correlation as being useful. This was apparent in the analysis done each

year in the projects themselves, where several teams raised their particular problem to a general problem. We recommend being firm with the rules for interpretation of the correlations as established.

So what attitude can we take towards measurements and statistical analysis? According to Demarco [20] there is a mantra that runs through all books and articles about software measurements, something along the lines of "Look, here is yet another wonderful metric that could show you useful things about how your organisation or project is doing". In recent studies, where we were involved [5], we suggest that the scope of the measurements should be narrowed, and used as indicators—not as the only basis for decision making. Measurements probably benefit from being kept at a restricted level where only a few measures for general purposes are collected. And they should be used, as Demarco suggests [20], primarily for discovery purposes, and not for behaviour modification and steering purposes.

6. Conclusion

It is very important to emphasise that this study focused on preserving and communicating explicit measurements, for using this general data for estimating and planning in future projects. This study does not confirm that generalised measurements will pay off by adding significant value to knowledge management in organisations. The result indicates that the great majority of measurements contain several uncertainties. Additionally, it is very hard to ensure that the processes are used uniformly. We have only managed to extract three planning constants that could be used for estimating and planning software projects. Good analysis is difficult because of all the uncertainty factors such as requirement changes, late hardware deliveries, and unplanned technical changes influence every single measurement differently.

We do believe that there are occasions where measurements fulfil an important purpose as an indicator for action. The measurement study regarding analysis of faults that have been presented in this paper is one good example. The measurements could, however, not be interpreted without a detailed analysis, primarily performed by discussions, in order to understand the different contexts. The study gave us a base for improvement, and the general proposals for avoiding faults extracted from the detailed analysis have added great value to our work.

The statistical study resulted in great scepticism regarding the use of the statistical calculations in the Software Engineering field. All the uncertainty factors, of course, also affect the statistical analysis. The seminars held were necessary for interpreting and understanding the measures. When we analysed the statistics in more detail, we found it impossible to draw general conclusions. Either the correlations were obvious or there were too many difficulties associated with understanding the different contexts.

References

[1] J. Liebowitz, "*Key Ingredients to the Success of an Organization's Knowledge Management Strategy*", Journal of Knowledge and Process Management, Vol. 6, No. 1, 1999, pp 37-40

[2] I. Nonaka, H. Takeuchi, "*The Knowledge-Creating Company*", Oxford University Press, 1995

[3] G. Bhatt, "*Managing Knowledge through People*", Journal of Knowledge and Process Management, Vol. 5, No. 3, 1998, pp 165-171

[4] V. Basili, G. Caldiera, H. D. Rombach, "*Experience Factory*", Encyclopaedia of Software Engineering Vol. 2 Editor: J. Marciniak, John Wiley and Sons Inc., 1994, pp 528-532

[5] C. Johansson, P. Hall, M. Coquard, "*'Talk to Paula and Peter - They are Experienced' - The Experience Engine in a Nutshell*", in G. Ruhe, F. Bomarius (ed.): Learning Software Organization - Methodology and Applications. Springer-Verlag. Lecture Notes in Computer Science, Volume 1756 (appears 4th quarter 2000).

[6] M. Hansen, N. Nohria, T. Tierney, "*What's your Strategy for Managing Knowledge?*", Harvard Business Review March-April 1999, pp 106-116

[7] M. Paulk et al, "*The Capability Maturity Model: Guidelines for Improving the Software Process*", Addison-Wesley, 1995

[8] L. Ohlsson, C. Johansson, "*A Practice Driven Approach to Software Engineering Education*", IEEE Transactions on Education, Vol. 38, No 5, 1995, pp 291-295

[9] A. Burr, M. Owen, "*Statistical Methods for Software Quality*", Int. Thomson Computer Press, 1996

[10] W. S. Humphrey, "*Managing Technical People*", Addison-Wesley, 1997

[11] C. Johansson, L. Ohlsson, P. Molin, "*Teaching Object-orientation: From Semicolons to Frameworks*", Proceedings of Software Engineering in Higher Education, 1994, pp 367-374

[12] I. Sommerville, "*Software Engineering*", 5th edition, Addison-Wesley, 1995

[13] C. Jones, "*Applied Software Measurement: Assuring Productivity and Quality*", 2nd edition, McGraw Hill, 1996

[14] IEEE, "*A Standard classification for Software Anomalies*", IEEE Computer Society Press, 1993

[15] B. W. Boehm, "*Software Engineering Economics*", Prentice Hall, 1981

[16] N. E. Fenton, S. L. Pfleeger, "*Software Metrics, A Rigorous & Practical Approach*", 2nd Rev edition, International Thomson Computer Press, 1998

[17] W. S. Humphrey, "*Introduction to the Team Software Process*", Addison-Wesley, 1999

[18] Practical Software Measurement Support Center, "*The Official Guide to the Practical Software Measurement Method*", version 3.1a, July, 1998, http://www.psmsc.com/

[19] R. W. Wolverton, "*The Cost of Developing Large-Scale Software*", IEEE Trans. Computers, June 1974, pp 615-636

[20] T. Demarco, "*Why does Software Cost so Much? And other Puzzles of the Information Age*", Dorset House, 1995

[21] W. S. Humphrey, "*A Discipline for Software Engineering*", Addison-Wesley, 1995