Gathering Experience Knowledge from Iterative Software Development Processes

Jouni K Kokkoniemi
Department of Information Processing Science
P.O. BOX 3000, FIN-90014 University of Oulu, Finland
Jouni.kokkoniemi@oulu.fi

Abstract

This paper proposes that experience knowledge would be beneficial for iterative software development. In this paper, experience knowledge-based artifacts have been linked to Extreme Programming via peer reviews. Experience knowledge can be defined as human experiences documented with the object of exploiting them in the organization. Extreme Programming has been selected to represent iterative software development processes. Systematically gathering experiences would support building evidence for improving and teaching iterative software development processes. This paper presents one possible solution as to how to put experience knowledge in practice in the context of iterative software development processes. It also reports results from projects where experience knowledge has been studied. The main findings of this paper are two ways in which experience knowledge-based artifacts can be used with iterative software development, and that peer reviews are a good way to collect and treat experience knowledge-based artifacts.

1. Introduction

There can be found several different kinds of iterative software development models nowadays. Agile techniques have become probably the most popular representative of such models. However, many software development models that have iterative and incremental features were already developed years ago, for example, evolutionary software development [32, 16], rapid prototyping [14], spiral model [10], OMT++ [1], and Rational Unified Process [22, 31].

In this paper, Extreme Programming (XP) [6] has been selected to represent iterative software development approaches. At the beginning of the paper, it is reasonable to say that Extreme Programming differs considerably from the methods mentioned previously. Iterations of Extreme Programming are significantly shorter and Extreme Programming can be classified as an agile technique [3].

Agile techniques apply the principles of light-weight production to the overall software lifecycle. These techniques usually give a set of practices and procedures that support either the whole software lifecycle or parts of it. Different agile methods focus on different aspects of the software development lifecycle or process [3]. For example, Extreme Programming is a collection of principles and practices that aims at enabling successful software development despite vague or constantly changing requirements in small and medium sized teams.

One common denominator of all agile techniques is communication and collaboration between the customer, project team, and other project stakeholders [7]. Short cycles, user involvement and integrated unit testing are used to avoid documentation overhead and ensure product quality. This has led to a reduction in the significance of peer reviews [39]. It should also be noted that agile techniques are suggested to be well suited for small projects with low criticality and changing requirements [11]. Based on what has been stated, it is true that the significance of peer reviews is less in small projects than in bigger ones.

Experience knowledge (EK) is an organizational memory-based concept, which is mainly based on human experiences [25]. It has taken features from the 'knowledge management of software engineering lessons learned' method presented by Birk and Tauz [9], the 'experience factory' method presented by Basili et al. [8], and the work of Nonaka and Takeuchi [30]. Ideas concerning experience knowledge have been evaluated and observed to be fit for use when studying inspections and inspection processes in

1530-1605/08 $25.00 © 2008 IEEE

partner companies [27, 28]. In this paper, experience knowledge-based artifacts have been linked to Extreme Programming via peer reviews.

The main purpose of this paper is to show how experience knowledge-based artifacts can be linked to software development processes. The objective of this paper is to answer the research question: how can experience knowledge be put into practice in the context of iterative software development processes? The original focus of this research was to investigate how experience knowledge-based artifacts can be generated [27] and how experiences can be gathered in the context of software development processes. The use of experience knowledge-based artifacts has been presented earlier and the use of experiences in process enactment and improvement has been presented previously by Kokkoniemi [25, 26, 28].

This paper will briefly present the selected agile technique, Extreme Programming. Then the paper presents the concept of experience knowledge and draws attention to potential areas of Extreme Programming in which experience knowledge-based artifacts can be used. Finally, the paper presents an example of how experience knowledge can be utilized via peer reviews.

2. Agile techniques and XP

Numerous agile techniques have emerged since the advent of the agile manifesto [7], each with different support for the development lifecycle phases. Abrahamsson et al. [4] discuss the evolution of agile techniques and analyze current methods for their use in supporting management, defined processes, and concrete practices throughout the phases of the software lifecycle.

Agile principles promote the delivery of software in small increments and iterations, which enables frequent progress checks and provides the opportunity to refine the project's goals in line with the customer's desires. Self-adaptation is also promoted at both the project and the process level [29]. Agile development methods are particularly suitable for projects in which the requirements are unclear and likely to change. One definition of agility is stated as "the ability to both create and respond to change in order to profit in a turbulent business environment." This is probably the primary advantage of agile approaches over plan-driven approaches [20].

Extreme Programming is probably the best known of the agile techniques. It can be described as a collection of practices and principles uncovered over recent decades. Extreme Programming emphasizes communication and collaboration within and between business and engineering teams. The impact of executing the practices, which strengthen each other, should negate the lack of traditional quality assurance activities. [6]

The values of Extreme Programming are communication, simplicity, feedback and courage, and these are further described through the principles rapid feedback, assumption of simplicity, incremental change, embracing change and quality work. Beck [6] suggests that most of the problems in software development arise due to lack of communication between project stakeholders. This is avoided through the execution of practices that concentrate on communication. Rapid feedback, incremental change and embracing change describe the possibilities for adaptation and learning provided by minimizing the time between providing a solution and receiving an opinion on it. This encourages the team to create quick and simple solutions, which can later be adjusted to fit situations that are more demanding. In summary, Extreme Programming primarily adapts software engineering practices in a way that promotes communication and co-operation within the project group and with the customer. It also provides a means for covering the common software engineering support processes, such as risk management and quality assurance, implicitly through its practices, instead of defining and executing them explicitly. Extreme Programming tracks the progress of the project and iterations through the completion of tasks and user stories that comply to acceptance test criteria.

Extreme Programming has received both positive and negative attention. The fact that it is a people-centric lightweight process with a focus on communication is seen as a positive side. However, certain of its practices, such as pair programming, refactoring and test-driven development, are the subject of much debate. Though there are some promising results [40, 41], proof as to whether these practices can deliver what they promise is scarce. There are also doubts concerning its application to critical systems [12, 23]. Suggested negative points include the minor amount of documentation causing maintenance problems, and the method's scalability in distributed or larger projects. The minor amount of documentation has also led to a reduction in the significance of peer reviews [19]. This is not usually a major problem, but when projects and the architecture of software increase, it can be.

3. Experience knowledge

Experience knowledge is an organizational memory-based concept, which is mainly based on human experiences [25]. It has taken features from the

'knowledge management of software engineering lessons learned' method presented by Birk and Tauz [9], the 'experience factory' method presented by Basili et al. [8], and the work of Nonaka and Takeuchi [30]. Ideas concerning experience knowledge have been evaluated and observed to be fit for use when studying inspections and inspection processes in partner companies [27, 28].

During the study of knowledge management in the context of inspections, an organizational memory-based concept, experience knowledge, has been defined [25]. It is based mainly on human experiences – with experience defined as "an event or occurrence, which leaves an impression on one" [13]. This definition is quite a general one and contains all possible experiences. On a general level, it has been concluded that a particular experience changes from the form of experience to that of experience knowledge when it is acquired, stored and shared systematically. Experience knowledge can be compared to experiential knowledge (Figure 1), as treated by Poikela [34] and Poikela [35], for example.

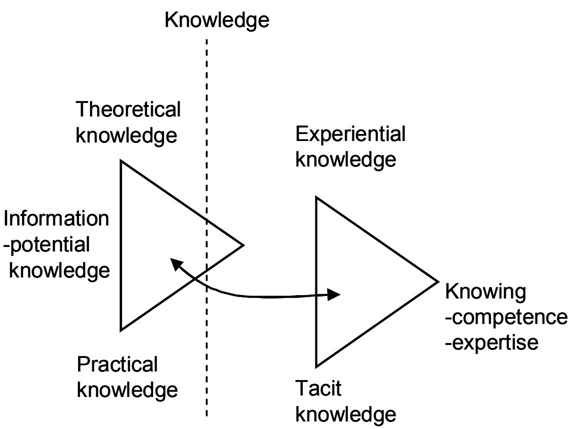

Figure 1. Information, knowledge, knowing [35].

Figure 1 illustrates how information changes into knowing through the transformation process of knowing. Information can be "whatever knowledge" encountered by any sense of the individual. In contrast, theoretical knowledge is conceptualized and symbolized information that is not meaningful until it is processed mentally. Practical knowledge is tangible and observable, like machines and organizations created by humans or objects of nature (like an ant hill). Theoretical knowledge as well as practical knowledge requires processing because objects and organisms cannot be understood without observations, concepts and experiments. [35]

Figure 1 also describes the contextual and chronological transition of learning from education to working life. This involves learning by practicing for working, socializing in a work community and identifying a profession. Learning at work is only partially conscious processing of knowledge. Access to tacit knowledge is increasingly becoming more important. This know-how of tacit knowledge must be shared with novices as it is the best way of supporting their learning and professional development. Knowing must also be shared with the educational system in order to organize experiences of workplace learning during education. [35]

In general, experiences are indicated as tacit knowledge and experience knowledge as explicit knowledge. A model for generating experience knowledge-based artifacts [27] has the main purpose of converting experiences to experience knowledge. The model helps an individual to perceive and understand possible purposes or values of a particular experience that he/she can utilize in software development.

One might ask how experience knowledge is different from other forms of explicit knowledge. Is it different from the concept of document management systems or version control, for example? Normally, document management systems or version control manage documents. If the contents of a document are changed, the document gets a new version number or the document is preserved in the document management system. One can tell, therefore, from a new version number, that the contents of the document have been changed. In the case of experience knowledge, the generated artifacts are the result of a process that converts experiences (tacit knowledge) into experience knowledge (explicit knowledge). When an artifact has been generated it is reasonable to use some kind of version control because the completed artifact needs maintenance and updating, just like any other document [26].

Three different phases relating to experience knowledge have been observed. These phases have been named 'recognition', 'preservation' and 'exploitation' [25]. In the recognition phase, a person perceives and understands possible purposes or values of a particular experience. It is not necessary to make a note of the experience at this stage because he/she may still wish to evaluate it. Once he/she can be sure of the value of the experience, it may be profitable to preserve it. It is also possible to preserve experiences considered to be of possible value in the future.

The fact that the medium of preservation can be of any form has brought about the conclusion that human experiences can be changed into the form of experience knowledge. One can use this preservation medium systematically. It is possible for a person to share their experiences while training new workers,

for example – preserving experiences in this way could be problematic if, for instance, the person with this particular experience changes jobs. In any case, this kind of experience knowledge repository is not in common use. The concession has been made that the human mind can be a repository, although if one wants to exploit experiences effectively one must use more solid ways of storing these experiences.

It is worth noting that, while a person's experiences change into the form of experience knowledge, the original experience stays in the human mind. Thus, while the person may forget certain initial experiences, when needed, experience can return in the form of the experience knowledge. This being the case, only a small fraction of experiences can be exploited as experience knowledge. While the majority of experiences can be considered tacit, experience knowledge is considered explicit.

In the exploitation phase, experiences are preserved in a solid form that is in public use. One possible solution in general is to use checklists or, in the case of Extreme Programming, "coding standards" or "own rules" [6], for example. These have been referred to as experience knowledge-based artifacts in the context of experience knowledge.

When considering some process in the context of knowledge management it is impossible to overlook certain details. It is possible to acquire quite a lot of data from the process and the participants in the process. It is also true that part of the collected data can be directly classified as experience knowledge (data related to human experiences) and part as process data (data that is based on a process, like metrics). In the case of an agile project, it is possible that one has collected information of the "lessons learned" type; this would be considered directly as experience knowledge, but collected metrics are not. However, in the context of experience knowledge they can both be related to experiences. The process data is not directly linked to human experiences but it is possible to collect metrics that may cause individual experiences. In any case, peer reviews are a good way to collect and treat experience knowledge-based artifacts.

4. Utilization of EK in the context of XP

The core agile values promote communication and collaboration, and in small teams, direct "face-to-face" communication is seen as an efficient and favorable way to disseminate information. Despite this, there can be situations in which mainly experience knowledge can be utilized. Because of the speed of software development, communication and collaboration are important, and it is important to collect and transfer the experiences gained from the process. This section presents potential experience knowledge-based artifact candidates within Extreme Programming found when checking over XP projects in the University of Oulu and in VTT Technical Research Centre of Finland, and discussed with experienced personnel from both organizations. In addition, this section presents how the gathered metrics (presented in Hulkko & Abrahamsson [21] and Abrahamsson & Koskela [2], for example) can be offered in an effective way for the exploitation of experience knowledge.

4.1. Experience knowledge-based artifacts

One interesting question concerns the kind of information that has been treated within Extreme Programming, and in what kind of artifacts this information has appeared. Extreme Programming seeks to minimize documentation [4, 19], so it can be assumed that there are not many mainly document-based experience knowledge-based artifacts within the compass of Extreme Programming. However, the study of Extreme Programming has found candidates that can act as experience knowledge-based artifacts [6, 3].

Possible experience knowledge-based artifacts identified by the practices of Extreme Programming include the following: user stories, new versions, metaphors, shared stories, test case code, coding standards, and own rules. These findings have not been examined more deeply in this paper because there are other interesting findings that have been made when examining Extreme Programming with experience knowledge.

The same kind of artifact candidates as those presented by Beck [6] can naturally be found among the different phases of Extreme Programming, but other noteworthy observations can also be made. In the exploration phase of Extreme Programming, tools, technologies and practices have been mentioned, for example, and one needs some kind of instructions to use and understand them. Although instructions are not generated in the same way as the experience knowledge-based artifacts mentioned [25, 26, 27], they can act in similar ways.

In the exploration phase of Extreme Programming, prototype architecture has also been mentioned ("build a few prototype architectures to help decide on the actual architecture to be implemented" in [6]). One example of a possible way to help building prototype architectures is the use of design patterns [18]. Design patterns can be used to solve repeatedly occurring design problems, and they display all the

characteristics of experience knowledge-based artifacts. Alghren and Markkula [5] have also arrived at the same kind of conclusion. In design patterns, experiences and knowledge as to how to solve problems has been preserved. However, this does not define design patterns accurately, as has been shown by Gamma et al. [18]. Agile design patterns can be generated in the company that uses them. They have the same structure as real design patterns but they are tailored for the target company, including experiences and knowledge from within it.

On further consideration, experience knowledge-based artifacts have been proved to support, in a certain sense, both knowledge gathering and knowledge transfer [27]. Figure 2 presents one possible way to use peer reviews when dealing with treating experience knowledge. The main point is that peer reviews are located between iterations even if there can be peer reviews within iterations. It is also practical to select some informal review method, such as peer deskcheck or passaround [39], for example.

The member of the project group gets direct experiences from iteration because he/she has been working on and has prepared the release. At the same time, he/she learns about new issues (1). Even if the amount of documentation is minor (2), it is profitable to review documentation. The review can be informal but it can support both quality-based and experience knowledge-based aspects of a project. If divergences in experiences have been observed, it is easy for the project member to compare metric data to experiences from the previous iteration, because of the

shortness of the cycle (3). The member of the project group will probably be able to remember the last two weeks, over which the iteration has lasted. This can create experiences that can later be transformed into experience knowledge form.

The reviewers are usually project members and customer representatives, and they can use tools to support their work. Support tools can include, for example, checklists or rules that guide software development. Both of these tools can act like experience knowledge-based artifacts.

The project member then continues to the next iteration (4). Issues adopted from the previous iteration are still in his or her mind, but the recognition of experiences can take some time. Therefore, it is possible that experiences gathered in the previous iteration will be documented within this iteration or within some later iteration.

It is profitable to document experiences for possible later use (5). If this document is in general use, as are all other documents in Extreme Programming ("Collective ownership"), it can be regarded as an experience knowledge-based artifact. For example, at group level this may take the form of some document in which new experiences are collected, or at individual level it may be some kind of diary or checklist. Diaries, etc. can be changed into experience knowledge-based artifact form when they become public.

The experience knowledge-based artifact may also re-enter the next iteration or iterations (6). This can happen if project members have short memories, or new members join the project group, for example. The experience knowledge-based artifact acts (7) as the basis for a document in the next iteration. Within this iteration, the existing document will be supplemented with new experiences. The experience knowledge-based artifact generated based on some iteration can also be used in some future project (8).

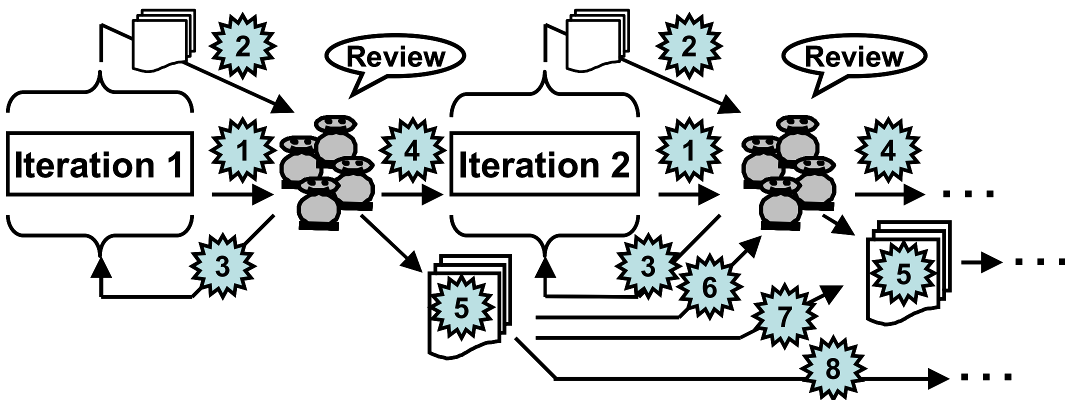

Figure 2. Peer reviews in the context of XP when treating experience knowledge.

Because of the short length of releases, it is important to concentrate on the most important elements. So there can be found at least one moderately well studied experience knowledge-based artifact, which is not generally used with Extreme Programming but which is able to support this requirement. Checklists have most commonly been regarded as an inspection tool [17], although on a few rare occasions they have been connected to other purposes in software engineering. In research, checklists have been studied, not only for their main purpose of finding defects when inspecting a software product, but also in collecting experiences from companies. It is also true that checklists can be used in every phase of software development. In the case of Extreme Programming, the checklist can be defined as a list of questions that guides participants to pay attention to the right things. The general problem is that although there is plenty of information for every layer of Extreme Programming, everybody makes their own decisions, and also their own "pioneer mistakes". One possible solution is for there to be 'design guidelines', 'coding checklists' and 'checklists for verification'. Separate checklists can be generated for every layer of Extreme Programming. The 'design guidelines' direct design, 'coding checklists' direct coding and 'checklists for verification' confirm quality.

These artifacts are only a few examples that can be used with Extreme Programming. One serious question is how they can be used without compromising the fundamental concept of agility. Another is how best to arrange the gathering, saving, use, and maintenance of checklists in the context of agile software development. The previous questions refer to discussions with agile software developers who thought that agile techniques do not need any new features. For example checklists or reviews seem to be too rigid to adapt to the process [19].

4.2. Process data

Gathering metrics within agile techniques has been proved to be profitable [2]. It has also been shown that process data can be one source of experience knowledge [27]. Figure 3 shows one possible way that the process data can be transformed into experience knowledge in the context of iterative software development. In Figure 3 the process (presented in Figure 2) has been extended with postmortem reviews [15, 36] whose modes of action give support to the transformation of process data and help participants to become conscious of their experiences linked to the metric data.

The member of the project group gets direct experiences from iteration because he/she has been working on and has prepared the release. At the same time, he or she learns about new issues (1). Metrics have been gathered during the iteration (2). This metric data must be stored somehow (for example, in a metrics repository).

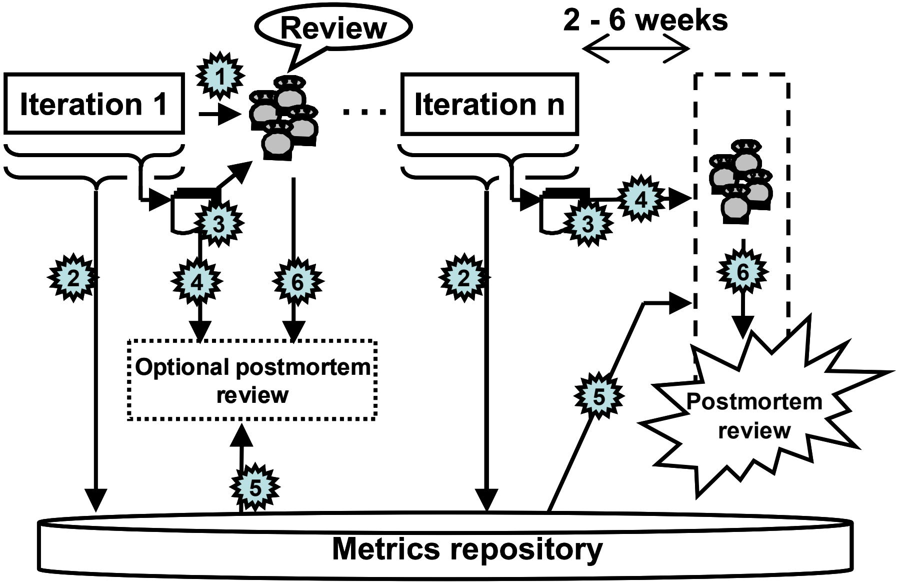

Figure 3. Transforming process data into experience knowledge.

3 depicts documents implemented during iteration, and arrow 4 shows how documents implemented during iteration are taken into an optional postmortem review. Collected metrics data are also taken into the postmortem review (5), as are the experiences of the participants of iteration (6). Postmortem reviews can be replaced by some other review practice, project retrospectives [24] for example, or by some light-weight process improvement practice, a "third hour" meeting [17], for example.

Figure 3 depicts a situation in which an iterative process is in action. The process could be Extreme Programming or some other iterative software development process. The rapid cycle of iterations and the gathering of metrics means that if a member of the project group has the chance to become familiar with the metrics collected (some resources must be allocated to this) concerning not only themselves but the whole project group, they can draw conclusions about their own activities during the iteration. This means that they can perhaps gain new experiences that can be changed into experience knowledge form. The distribution of metrics data to project members can be organized in many ways. One possibility is peer review, and another the use of an intranet, as has been mentioned in Pehkonen [33].

Dingsøyr [15] has introduced practical methods for the harvesting of experience from projects that have either been completed or have finished a major activity or phase. In bigger projects, there can also be postmortem reviews as the project goes on (optional postmortem reviews in Figure 3). The main reason that postmortem reviews have been carried out during an ongoing project is that they support knowledge management and knowledge transfer from metrics data to experiences and from experiences to experience knowledge.

It is worth noting that all process data (metrics) and experience knowledge-based artifacts can be exploited in future projects. The document generated (in Figure 2) can be used when new employees join the company, for example. It can be informal in format, or formatted as a design pattern. In fact, this practice has been preliminarily evaluated with agile techniques, but it works in general with iterative processes.

4.3. Knowledge creation

Nonaka and Takeuchi [30] explain the spiral of organizational knowledge creation as depicted in Figure 4. Different phases of the knowledge creation spiral can be linked to experience knowledge and to Figures 1 and 2. The basic ideas of this section have been adopted from the research of Tervonen and Kerola [38], but also from the results of our research [27].

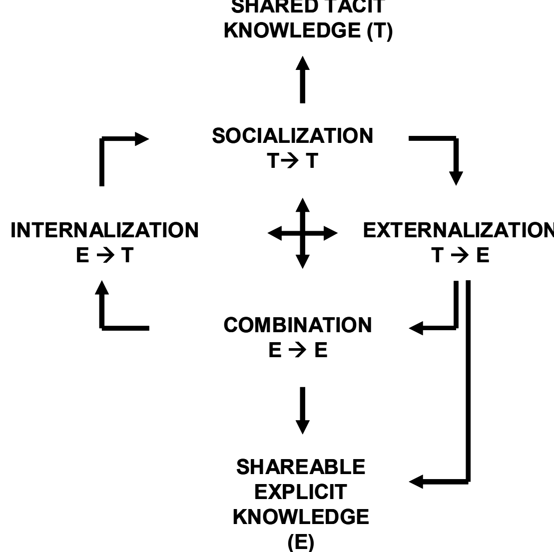

Figure 4. Four modes of knowledge creation and conversion.

Socialization is the process of sharing experiences and thereby creating tacit knowledge such as mental models and technical skills. An individual can acquire tacit knowledge directly from others without using language, i.e. through observation, imitation, and practice. This socialization happens when working together in a project group during iterations, and when participating in reviews. This mode serves the recognition phase of experience knowledge. In the recognition phase, a person perceives and understands possible purposes or values of a particular experience. It usually means that participants share experiences of the implemented software product, software development process and review process. In this case treated knowledge is shared tacit knowledge for software development.

Externalization is the process of articulating tacit knowledge in the form of explicit concepts, in the form of metaphors, analogies, concepts, hypotheses or models. Once explicit concepts (artifacts or knowledge profitable to serve in this case) are created, they can be used in modeling. Models are usually generated from metaphors when new concepts are created in the business context. In this mode a participant or participants have made the decision that some experiences are worth saving in some format (the preservation phase of experience knowledge). In other words, tacit knowledge has been organized into explicit form. It is also possible to preserve

experiences considered to be of possible value in the future.

Combination is the process of systemizing concepts into a knowledge system. This mode of knowledge conversion involves combining different bodies of explicit knowledge. Individuals exchange and combine knowledge through such media as documents, meetings, and collected metrics. Reconfiguration of existing explicit information through sorting, adding, combining and categorizing can lead to new knowledge which can be used in the next iteration. In this mode, shareable explicit knowledge for software development has been captured, organized and displayed. This mode mainly serves the exploitation phase of experience knowledge.

Internalization is the process of embodying explicit knowledge in tacit knowledge. It is closely related to 'learning by doing' or 're-experiencing' through recorded artifacts. When experiences are internalized through socialization, externalization, and combination into the individual's tacit knowledge bases in the form of shared mental models and technical know-how, these become valuable assets. This mode serves both the exploitation and recognition phases of extreme programming, and it usually happens during iterations.

For organizational knowledge creation to take place, however, the tacit knowledge accumulated at the individual level needs to be socialized with other organizational members, thereby starting a new spiral of knowledge creation. For explicit knowledge to become tacit, it helps if the knowledge is verbalized or diagrammed into documents, manuals or oral stories. Documents or manuals facilitate the transfer of explicit knowledge to other people, thereby helping them appreciate the experiences of others indirectly (i.e. to 're-experience' them).

5. Conclusion

This paper has presented some ideas as to how experience knowledge can be treated in iterative software development. The results presented were based on the Tarjous research project [37], observations made when checking over XP projects in the University of Oulu and in VTT Technical Research Centre of Finland, and discussed with experienced personnel from both organizations, and the preliminary results of a few studies that have not yet been published. In this paper, Extreme Programming was selected to represent iterative software development approaches. One possible way to utilize experience knowledge with Extreme Programming is through peer reviews. This paper first presented some arguments as to why it is reasonable and profitable to link Extreme Programming and experience knowledge together. Secondly, it presented the experience knowledge-based artifact candidates observed when studying Extreme Programming. Potential experience knowledge-based artifact candidates within Extreme Programming include user stories, coding standards, and own rules.

The main observation of this paper shows two ways in which experience knowledge-based artifacts can be used with iterative software development. Another important observation is that peer reviews are a good way to collect and treat experience knowledge-based artifacts. Two observed experience knowledge-based artifacts are design patterns and checklists, and we have made observations concerning how to transform process data into experience knowledge in Extreme Programming (or any other iterative process in which metrics have been collected).

It has been shown that with an iterative software development process there is a great need to collect and transfer knowledge. In rapid software development, it is also important to direct participants' attention to the right, significant issues. One suggested artifact that directly supports these requirements is checklists. Another noteworthy artifact that can support such requirements is design patterns. It is also true that systematically gathering experiences would support building evidence for improving and teaching iterative software development methods, and that peer reviews are a good way to collect experiences.

The thoughts presented here have been evaluated while checking over XP projects in the University of Oulu and in VTT Technical Research Centre of Finland, and have been discussed with experienced personnel from both organizations. Participants in agile projects in both organizations have understood the artifacts and reviews of work presented in this paper, but they do not want to add new tasks in agile methods. The research is to be continued with experience knowledge, Extreme Programming, and with other iterative software development processes.

6. References

[1] Aalto, J.M. and Jaaksi, A., "Object-Oriented Development of Interactive Systems with OMT++", Proceedings on TOOLS 14 (Technology of Object-Oriented Languages & Systems), Prentice-Hall, 1994, pp. 205–218.

[2] Abrahamsson, P. and Koskela, J., "Extreme programming: a survey of empirical data from a controlled case study", Proceedings on Empirical Software Engineering, 2004, pp. 73–82.

[3] Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J., Agile Software Development Methods – Review and analysis, VTT Publications 478, Otamedia Oy, Espoo, Finland, 2002.

[4] Abrahamsson, P., Warsta, J., Siponen, M.T., and Ronkainen, J., "New Directions on Agile Methods: A Comparative Analysis", 25th International Conference on Software Engineering (ICSE'03), IEEE Computer Society; 2003, pp. 244–254.

[5] Alghren, R. and Markkula, J., "Design Patterns and Organizational Memory in Mobile Application Development", Proceedings on PROFES 2005, LNCS 3547, Springer-Verlag, Berlin Heidelberg, 2005, pp. 143–156.

[6] Beck, K., Extreme Programming Explained: Embrace Change, Addison-Wesley, 2000.

[7] Beck, K. and Beedle, M., Manifesto for Agile Software Development, http://agilemanifesto.org/, 2001.

[8] Basili, V., Caldiera, G., and Rombach, H.D., "The Experience Factory", Encyclopedia of Software Engineering, vol 1, John Wiley & Sons, 1994, pp. 469–476.

[9] Birk, A. and Tauz, C., "Knowledge Management of Software Engineering Lessons Learned", Proceedings of the 12th Conference on Software Engineering and Knowledge Engineering, 1998, pp. 24–31.

[10] Boehm, B.W., "A Spiral Model of Software Development and Enhancement", IEEE Computer, 21(5), 1988, pp. 61–72.

[11] Boehm, B.W. and Turner, R., Balancing Agility and Discipline: A Guide for the Perplexed, Addison-Wesley Longman Publishing Co., 2003.

[12] Cohen, D. and Lindvall, M., Agile Software Development – State-of-the-art Report, Fraunhofer Center for Experimental Software Engineering and The University of Maryland., 2003.

[13] The Concise Oxford Dictionary, 12th edition, New York, Oxford University Press Inc, 1999.

[14] Curtis, W., Krasner, H., Shen, V., and Iscoe, N., "On Building Software Process Models under the Lamppost", proceedings of the 9th International Conference on Software Engineering, IEEE Computer Society Press, 1987, pp. 96–103.

[15] Dingsøyr, T., "Postmortem reviews: purpose and approaches in software engineering", Information and Software Technology, 47(5), 2005, pp. 293–303.

[16] Gilb, T., Principles of Software Engineering Management, Addison Wesley, 1988.

[17] Gilb, T. and Graham, D., Software Inspection, Wokingham, UK, Addison-Wesley, 1994.

[18] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, USA, 1995.

[19] Hedberg, H. and Iisakka, J., "Technical Reviews in Agile Development: Case Mobile-D™", Proceedings of Quality Software, (QSIC 2006), 2006, pp. 347–353.

[20] Highsmith, J., Agile Project Management, Addison-Wesley, Boston, MA, 2004.

[21] Hulkko, H. and Abrahamsson, P., "A multiple case study on the impact of pair programming on product quality", Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005, pp. 495–504.

[22] Jacobson, I., Booch, G., and Rumbaugh, J., The unified software development process, Addison-Wesley Longman Publishing Inc., Boston, MA, USA, 1999.

[23] Keefer, G., Extreme Programming Considered Harmful for Reliable Software Development, Proceedings of CONQUEST 2002, Nurenberg, Germany, 2002.

[24] Kerth, N.L., Project Retrospectives – A Handbook for Team Reviews, Dorset House Publishing, New York, USA, 2001.

[25] Kokkoniemi, J., "Checklists as a Tool for Collecting Experience Knowledge", Proceedings of IASTED International Conference of Information and Knowledge Sharing (IKS 2002), St. Thomas, Virgin Islands, USA, 2002, pp. 223–228.

[26] Kokkoniemi, J., "The Lifecycle of Checklists: A Knowledge Management-based Approach", Proceedings of the Third European Conference on Knowledge Management, Dublin, Ireland, 2002, pp. 376–386.

[27] Kokkoniemi, J., "A Preliminary Model for Generating Experience Knowledge Based Artifacts", Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06) Track 7, Kauai, Hawaii, USA, 2006, 153b

[28] Kokkoniemi, J., "Experiences from Generating Checklists". Proceedings of the IASTED International Conference on Knowledge Sharing and Collaborative Engineering (KSCE2006), St. Thomas, Virgin Islands, USA, 2006, pp. 51–56.

[29] Lindvall, M., Basili, V.R., Boehm, B.W., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L.A., and Zelkowitz, M.V., "Empirical Findings in Agile Methods", Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe, 2002, pp. 197–207.

[30] Nonaka, L. and Takeuchi, H., The Knowledge-Creating Company – How Japanese Companies Create the Dynamics of Innovation, Oxford Press, 1995.

[31] Krutchten, P., The Rational unified process: an introduction, Addison-Wesley, 1999.

[32] Lehman, M.M. and Belady, L.A., Program evolution: processes of software change Academic Press Professional Inc., San Diego, CA, USA, 1985.

[33] Pehkonen, A., Testauksen automatisointi ja organisointi testivetoisessa ohjelmisto-kehityksessä - Tapaus: Automaster Oy (in Finnish), University of Oulu, Department of Information Processing Science, Master's thesis, Oulu, Finland, 2005.

[34] Poikela, S., Ongelmaperustainen pedagogiikka ja tutorin osaaminen (in Finnish), Tampere University Press, 2003.

[35] Poikela, E., "Developing criteria for knowing and learning at work: towards context-based assessment", Journal of Workplace Learning, 16(5/6), 2004, pp. 267–274.

[36] Salo, O., Enabling Software Process Improvement in Agile Software Development Teams and Organisations, VTT Publications 618, Otamedia Oy, Espoo, Finland, 2006.

[37] Tervonen, I., Iisakka, J., Harjumaa, L., Kokkoniemi, J, and Hedberg, H., Reorganising Software Inspection: Challenges, Ideas and Preliminary Solutions (The final report of the TARJOUS research project), University of Oulu, Department of Information Processing Science, research papers series A 30, Oulu, Finland, 2000.

[38] Tervonen, I. and Kerola, P., "Towards deeper co-understanding of software quality", Information and Software Technology 39, 1998, pp. 995–1003.

[39] Wiegers, K.E., Peer Reviews in Software: A Practical Guide, Addison-Wesley, Boston, USA, 2002.

[40] Williams, L. and Kessler, R., Pair Programming Illuminated, Addison-Wesley Longman Publishing Co., Inc, 2002.

[41] Williams, L., Maximilien, E.M., and Vouk, M., "Test-Driven Development as a Defect-Reduction Practice", Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003, p. 34.