Knowledge Sharing: Agile Methods vs. Tayloristic Methods

Thomas Chau, Frank Maurer, Grigori Melnik
Department of Computer Science
University of Calgary
Calgary, Canada
{chauth,maurer,melnik}@cpsc.ucalgary.ca

Abstract

This paper presents a comparative analysis of knowledge sharing approaches of agile and Tayloristic (traditional) software development teams. Issues of knowledge creation, knowledge conversion and transfer, continuous learning, competence management and team composition are discussed. Experience repositories and other tools for knowledge dissemination are examined.

1. Introduction

Software engineering is a knowledge-intensive process encompassing requirements gathering, design, development, testing, deployment, maintenance, and project coordination and management activities. It is highly unlikely that all members of a development team possess all the knowledge required for the aforementioned activities. This underlies the need for knowledge sharing support to enable software organizations to (1) effectively share domain expertise between the customer and the development team; (2) identify the requirements of the software system; (3) capture non-externalised knowledge of the development team members; (4) bring together knowledge from distributed individuals to form a repository of organisational knowledge; (5) retain knowledge that would otherwise be lost due to the loss of experienced staff; and (6) improve organisational knowledge dissemination.

More traditional approaches, like the Waterfall model and its variances, facilitate knowledge sharing primarily through documentation. They also promote usage of role-based teams and detailed plans of entire software development lifecycle. The allocation of work specifies "not only *what* is to be done but *how* it is to be done and the *exact time* allowed for doing it" [24]. This shifts the focus from individuals and their creative abilities to the processes themselves. These traditional approaches are often referred to as "plan-driven" or "task-based". In contrary, agile methods emphasise and value individuals and interactions over processes [6]. When comparing agile and traditional methods, we prefer to use the term "Tayloristic approaches" when discussing the traditional methodologies. We believe that the latter should not be referred as "plan-driven", because agile methods are also plan-driven. In fact, we argue that agile methods may involve more planning activities than Tayloristic approaches but of shorter cycles (iterations). The term "task-based" should also be avoided as it points to the side effect of Tayloristic methods, rather than the cause.

Tayloristic methods heavily and rigorously use documentation for capturing knowledge gained in the activities of a software project lifecycle; ensuring product and process conformance to prior plans; supporting quality improvement initiatives; and satisfying legal regulations.

In contrast, agile methods suggest that most of the written documentation can be replaced by enhanced informal communications among team members internally and between the team and the customers with a stronger emphasis on tacit knowledge rather than explicit knowledge [21]. They argue that the cost of creating and updating documents against frequent changes in the requirements and source code often outweigh the benefits of documenting the system and domain knowledge in details. Having said that, agile methods do not completely leave out documentation, but rather promote self-documenting designs and self-describing code that conforms to coding standards and guidelines (either industry-wide or internal). Some agile methods endorse modeling as a standard for documentation (e.g. Agile Modeling), but others consider it to be too heavyweight (e.g. XP). But all of them agree that documentation should be *lean* and *mean* and there should be *just enough* of it.

Knowledge creation and sharing are crucial parts of both agile and Tayloristic software development processes. For completeness, we include a short overview of basic concepts from the area of knowledge management in section 2. Section 3 covers some background on agile development approaches. In section 4, we compare how knowledge sharing is handled by both

agile and Tayloristic methods in the following dimensions: documentation, capture of requirements and domain knowledge, training, competence management, team composition, continuous learning, and knowledge repositories.

2. Knowledge Management Background

Knowledge management is a discipline that crosses many areas. It involves a variety of subjects such as economics, psychology, informatics, and technology. Hence, there exist various definitions of knowledge and knowledge management [1]. In this paper, we rely on the learning model proposed by Nonaka and Takeuchi [2].

This learning model categorizes knowledge into tacit and explicit forms. Based on these two forms of knowledge, the model differentiates four ways of transforming knowledge. They are socialization, externalization, internalization, and combination.

This categorization of knowledge and knowledge transformation helps to explain the different approaches Tayloristic and agile methods take in supporting knowledge sharing.

3. Agile Methods Background

In recent years, agile development methods have attracted much attention in the software development community. Many variations exist [7-11, 14-16], but all of them share the common principles and core values specified in the Agile Manifesto [6].

4. Knowledge Sharing Support in Agile and Tayloristic Methods

We will analyze the different strategies Tayloristic and agile methods take in supporting knowledge sharing in the following dimensions: documentation, capture of requirements and domain knowledge, training, competence management, team composition, continuous learning, and knowledge repositories.

4.1. Documentation

Common to all software development processes in any projects is the need to capture and share knowledge about the requirements and designs of the product, the development process, the customer's business domain, and the project status.

In Tayloristic development approaches like the Waterfall model and its variants (SDM, SSDM, SADM, Navigator, ForeFront, Method/1, Summit), most, if not all, of this knowledge is externalised to multitude of documents to ensure all possible requirements, design, development, and management issues are addressed and captured.

One advantage to this emphasis on knowledge externalisation is that it reduces the likelihood of loss of knowledge as a result of knowledge holders leaving the organisation. By externalising knowledge into explicit form, Tayloristic methods also enable distributed software teams to collaborate in a time- and space-independent manner. On the other hand, most of the knowledge in software engineering is tacit. Few of them can be made explicit and few of the explicit knowledge can be documented in details because software developers are often reluctant to do so due to tight time constraints and the huge effort they perceived is required for documenting what they know [4]. Even if most of the knowledge is documented, there is the issue of ensuring the documented knowledge is up-to-date.

For this reason, agile methods advocate *lean* and *mean* and "*just enough*" documentations. For instance, in Extreme Programming (XP), one of the agile development approaches, requirements knowledge is externalised to index cards [8]. Other agile methods, like Feature Driven Development (FDD) and Agile Modeling (AM), suggest domain knowledge and system design alternatives to be externalised in the form of models [10, 16]. AM in particular suggests this should be done only if the models facilitate better communication or understanding of the system. As such, models needs not be very detailed. In cases when detailed models are required as in FDD, sophisticated CASE tools are recommended to reduce the amount of manual effort required for generating and updating those models.

AM also recommends these models should be made public for the entire team to see (for example, by posting them on a wall of the work area). This helps facilitating knowledge distribution. The use of modeling standards is similar to the use of coding standards in XP. Both help to reduce knowledge transfer time by avoiding time-consuming debates of coding/modeling styles.

Scrum, another agile method, advocates "work-in-progress" as the only documentation unless the documentation will be used by others to create a vision (marketing) or to operate the system (user documentation) [14].

Although it is not stated, several agile methods (e.g. XP and Scrum) imply that explicit knowledge including designs and models should be collectively owned. Collective ownership can facilitate knowledge evolution in that anyone in the team can update the model when one notices it is outdated. If particular individuals own the models and no one else is permitted to update them, the model tends not to get updated since others may find it too burdensome to go through the approval process to update the models.

However, the fact that any team member can update the model does not guarantee the model is current. In fact, AM suggests to reduce the overhead of constantly ensuring the models are current, models should only be updated when the cost of using the outdated model outweighs the cost of updating the models. One implication of this is that it may hinder reuse of knowledge since the outdated model may be seen as too outdated for use. Furthermore, it is not stated how to determine when cost of using the outdated model outweighs the cost of updating them.

Compared to Tayloristic methods, there is significantly less documentation in agile methods. As less effort is needed to maintain fewer documents, this improves the probability that the documents can be kept up to date. To compensate for the reduction in documentation and other explicit knowledge, agile methods strongly encourage direct and frequent communication and collaboration whenever possible in order to tap the tacit knowledge within the team.

It is important to note that agile methods are often used for small projects that are localised and hence can survive without documentation. However, in a distributed or large organisation where face-to-face collaboration or communication is inconvenient, documentation may play a much more important role.

4.2. Requirements and Domain Knowledge

With respect to gathering requirements and domain knowledge, agile methods advocate strongly for active stakeholders and users participation through practices ranging from joint-application design (JAD) sessions and customer focus groups [7] to on-site customers [8]. Some agile methods extend this effort further to explicitly specified business (domain) study [9] or domain modeling sessions [10]. Although Tayloristic methods do not suggest any specific practices that support active stakeholder and user participation, some of the above requirements engineering techniques are also practiced in some Tayloristic projects.

All these practices facilitate collaboration between the customers and the development team in determining and planning system features to be implemented. System and/or domain knowledge is disseminated to the development team more effectively due to the frequent and close contact with the customers. Compared to approaches where only the business analyst discuss the requirements with the customers and delegates development tasks to developers, the above practices allows most of the development team to understand the system better through mandatory collaboration between the team and the customers. Through dialogue, individuals' mental models and skills are converted to common concepts and understanding. Sharing time and space together allows nourishing and energising this type of collective tacit knowledge.

Tayloristic processes differ from agile methods in that all requirements are captured before any design and development. A side effect of this is that the development team rarely interacts with the customers to gain any feedback on their understanding of the system requirements. This approach is efficient as long as system requirements remain stable till the project ends. However, there exist scenarios where rapid and constant changes to requirements are unavoidable. XP addresses this issue by having customer representatives working at the development team's site. This practice allows the developers to communicate directly with the customers throughout the development cycle. Consequently, system requirements can be acquired and clarified much faster. A limitation of this practice is that the customers and the developers need to be co-located. This is often not possible given the distributed nature of the workforce in the current business environment. There are, however, several studies looking at adapting agile methodologies for use in a distributed environment (see [22, 23]). They show that through the groupware, teams can communicate and collaborate on projects even if they are not co-located. This is not limited to synchronous communication only. Case studies provide evidence of virtual teams working even in different time zones [25].

Volatile system requirements can be attributed to the fact that customers generally do not know their real needs and wants until they see and use a functional component of the system. Capitalising on this phenomenon, most agile methods mandate small and frequent releases, which allow both the development team and the customers to have a better understanding of the system and allow customers to generate requirements that are fit for them. This is because the customers get to see working features more frequently. Working closely with the customer provides an opportunity for what Masao Maekawa calls "seamless co-experiencing". Customer's needs and the knowledge required to solve the problems are more tacit, and often customers find it hard to express it explicitly. So, by experiencing what customers are experiencing, developers actually get the knowledge required to solve the problems effectively and to avoid a common situation in the Tayloristic world when the customer declares that the large set of features presented at the end are not satisfying their needs. This happens either because of the disconnect of the development team and the customer or simply because customer's needs have changed since the requirements were originally captured and frozen in the requirements specification document.

4.3. Training

With regards to disseminating process and technical knowledge from experienced team members to novices in the team, Tayloristic and agile methods use different training mechanisms as well. While it is not stated, formal training sessions are commonly used in Tayloristic organizations to achieve the above objective. Agile methods, on the other hand, recommend informal practices (e.g. pair programming and pair rotation in case of XP). Pair programming involves two programmers working in front of one computer designing, coding, and testing the software together.

One advantage with formal training sessions is that training content and practices can be standardized and be applied consistently across multiple teams in a large organization but formal training sessions are expensive as they mean loss of valuable development time for both the trainers and the trainees.

Such problems, however, are not evident in pair programming. A practitioner reported that "since pairing is a part of daily life, no one has to take downtime to help out the new person. Much of the mundane technical training can be assimilated as part of the job" [12]. XP also recommends pairs be rotated in the entire development team. XP proponents cite these practices have the benefits of: decreased learning curve by 84% [11]; improved communication and coordination [11]; fostering a culture of knowledge sharing; and facilitate the sharing of tacit knowledge. Examples of tacit knowledge being shared include system knowledge, coding convention, design practices, and tool usage tricks. Developers tend not to document this knowledge and it is usually not explicitly taught through formal training. Study also indicates that pair programming together with regular meetings helps mitigate risks of knowledge loss due to attrition [13].

Informal training approaches like pair programming and pair rotation are not problem-free unfortunately. Training content may vary, or worse, conflict across different pairs. Assigning two people to work cooperatively as a pair is also an extremely tricky task. One may argue that pair programming constantly reduces the productivity of the experts as they need to train novice all the time and formal training is therefore less expensive. We believe that pair programming can be more expensive than formal training, or vice versa, depending on the circumstances. It should be possible to put in place a training infrastructure that has the benefits of both approaches.

4.4. Competence Management

Identifying what your staff knows or doesn't-know is known as competence management. Studies have shown that people are often not aware of knowledge holders that might be relevant to them [19].

To address this problem, agile methods suggest daily stand-up meeting during which each developer (or a pair) needs to present his/her work done since the last meeting [14]. Team members may also voice their enquiries during the meetings. Such presentation provides visibility of the presenter's work to fellow developers and project managers. Everyone in the team knows who has worked on or is knowledgeable about which parts of the system. They know whom to contact when they need to work on parts of the system that they are unfamiliar with.

While Tayloristic methods do not mandate any specific practice to deal with this issue, a common practice is to identify experts based on document authorship.

4.5. Trust and Care

As software development is a very social process, it is important to develop organisational and individual trust in the teams and also between the teams and the customer. Trusting other people (and their code) facilitates reusability and leads to more efficient knowledge generation and knowledge sharing. Through collective code ownership, stand-up meetings, onsite customer, and in case of XP, pair programming, agile methods promote and encourage mutual trust, respect and care among developers themselves and with respect to the Customer. The key of knowledge sharing here are the interactions among members of the teams which happen voluntarily, and not by an order from the headquarters.

4.6. Team Composition

In a large organization, it is often the case that different roles emerge. In Tayloristic teams, these different roles are grouped together as a number of role-based teams each of which contains members of the same role. In contrast, agile teams use cross-functional teams. Such a team draws together individuals performing all defined roles. Rotations from one role to another are common. It is also possible to have highly specialised experts (for example, security analysts and usability engineers) shared among several teams in an organisation.

One advantage to role-based teams is that teams whose work products are independent of each other can work in parallel as long as there is not much knowledge flow among the different functional sub-team. This is often seen in repeatable manufacturing-like processes [23]. However, in knowledge-intensive software development that demands information flow from different functional sub-teams, role-based teams tend to lead to islands of knowledge and difficulty in its sharing among all the teams. As hand-offs between teams usually

are based on document flow, the knowledge of one team that is required by the other team must be externalised and documented. Although reviews try to minimize the knowledge loss, externalisation and documentation processes cannot guarantee that all knowledge is captured and even if most of it was rigorously captured, there is still no guarantee or way to check its correctness till the project sign-off.

Cross-functional teams should be used to facilitate better knowledge transfer. This is especially the case for agile methods since they are recommended to be used where there is a lot of uncertainty and unknown knowledge about the domain and system requirements, and the technologies to be used are new and unexplored.

4.7. Continuous Learning

Continuous learning is supported by some agile methods in the form of retrospectives. Examples include Post-Sprint meetings [14], reflection workshops [15], post-iteration phases [9], and review phases [7]. These are in essence post-mortem reviews except that they are conducted not only at the end of a project but also during the project. Retrospectives facilitate learning of any success factors and obstacles of the current management and development process. In cases where team members face obstacles of the current process, such as for example stand-up meetings being too long, retrospectives provide the opportunity for these issues to be raised, discussed, and dealt with during the project rather than at the end of project. Retrospectives in agile methods, however, only facilitate intra-team learning. Together with other agile practices, they have no explicit support for inter-team learning within an organisation.

Tayloristic organizations also use retrospectives and they are often conducted after big milestones and at the end of projects. The duration of these big milestones are much longer than the iteration lengths in agile projects. Hence, fewer retrospectives are performed in Tayloristic projects. Unlike agile teams, Tayloristic teams support continuously learning not only at the project team but also at the organization levels. The latter is achieved by having a separate process group that analyzes experiences from different project teams and refines the standard development process which all project teams in the organization need to conform to. This is in conflict with the agile principles. There is recent work that attempts to use a tool-oriented approach to generate adaptive development methodologies, however, effectiveness of such approach is still unknown [18].

4.8. Knowledge Repositories

As mentioned before, Tayloristic methods rely heavily on explicit knowledge to ensure conformance to prior plans and externalisation to mitigate knowledge loss. In most of these organisations, their existing infrastructure to facilitate the capture and sharing of knowledge is based on the Experience Factory concept [3] and experience repositories in particular [4]. Current implementations of the experience repositories range from a mere document repository to an expert finder for expert identification to Process-Centered Software Engineering Environments (PSEEs) for providing context-sensitive knowledge. [4, 5]

The Experience Factory strongly advocates reuse of previous experience and having a centralised team responsible for repository maintenance. The former reinforces continuous learning on both team and organisation levels. The latter has the benefit of making knowledge that was gained by particular project teams accessible to the entire organisation.

On the other hand, critics argue that the repository-only approach does not address how well users internalise and use this explicit knowledge or how users' tacit knowledge is managed [20]. They claim that learning or the internalisation of explicit knowledge is a social process. One does not learn alone but learns mainly through tacit knowledge gained from interactions with others. Furthermore, tacit knowledge is often difficult to be externalised into a repository. A repository by itself also does not support communication or collaboration among people. Although these criticisms can be addressed by expert finders or PSEEs, there are also problems with these two approaches. Expert finder has the potential problem of the profiles of the knowledge holder being outdated or overstated. Although some of PSEEs support dynamic changes in the users' tasks, majority of the modern PSEEs remain static and less adapting.

We believe that due to the high complexity of the software process in general, it is hard to create and even more difficult to effectively maintain the experience repository. The operation support cost may overweight the benefits of such experience repository. Hence, tools like the Wiki Web may be more appropriate as a knowledge repository for agile teams [17]. Unlike implementations of the Experience Factory concepts, the responsibility and capability of maintaining the content stored in a Wiki Web is decentralised to everyone in the team. The fact that anyone can update any type of content posted on a Wiki Web at anytime without undergoing a rigorous review-approval process normally associated with Experience Factories eases the burden of maintaining the knowledge repository. The open and informal nature of the tool allows the team to control the amount and details of knowledge to be externalised. On the other hand, the informal nature of the tool places great responsibilities on the shoulders of every team member to ensure the quality of the content stored in the repository. The informal nature of the tool may also poses difficulty in retrieving information in an efficient manner.

5. Conclusion

Tayloristic development approaches support knowledge sharing primarily by explicit knowledge externalised in documents or repositories. The Experience Factory concept of reusing previous project experiences and a centralised knowledge management organisation provides the infrastructure necessary in supporting continuous learning at the project team and organisation levels. On the other hand, its main drawbacks are that it does not address issues of how well users internalise explicit knowledge and the sharing of tacit knowledge that is not externalised.

Agile development approaches rely heavily on socialisation through communication and collaboration to access and share tacit knowledge within the project team. When externalisation and internalisation are used to transfer knowledge, all agile methods suggest that they should be supported by close communication and collaboration. All agile methods involve the customers directly in acquiring requirements and domain knowledge. An iterative development approach is used to provide rapid feedback and continuous learning between the customers and the development team. To facilitate learning among developers, agile methods use daily/weekly stand-up meetings, pair programming, pair rotation and collective ownership. The use of retrospectives also supports continuous learning at a project team level. Agile methods' emphasis on people, communities of practice, communication, and collaboration excels in facilitating the practice of sharing tacit knowledge at a team level. They also foster a team culture of knowledge sharing, mutual trust and care. In addition, tools like Wiki enable easy and effective sharing of explicit knowledge.

6. References

[1] M. Alavi, D. Leidner, "Knowledge Management Systems: Issues, Challenges, and Benefits", Communications for the Association for Information Systems, Vol. 1, Article 7, 1999.

[2] I. Nonaka, "A Dynamic Theory of Organizational Knowledge Creation", Organization Science, Vol. 5, No. 1, 1994, pp. 14-37.

[3] V. Basili, G. Caldiera, H. D. Romback, "Experience Factory", In Encyclopedia of Software Engineering, Vol. 1, John Wiley & Sons, 1994, pp. 476-496.

[4] I. Rus, M. Lindvall, "Knowledge Management in Software Engineering", IEEE Software, May-June 2002

[5] P. Garg, M. Jazayeri, "Process-centered Software Engineering Environments", IEEE Computer Society Press, 1996.

[6] Agile Manifesto. http://agilemanifesto.org

[7] J. Highsmith, K. Orr, Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, Addison Wesley, 2000.

[8] K. Beck, Extreme Programming Explained: Embrace Change, Addison Wesley, 1999.

[9] J. Stapleton, DSDM Dynamic Systems Development Method: The Method in Practice, Addison Wesley, 1997.

[10] P. Coad, J. de Luca, E. Lefebvre, Java Modeling Color with UML: Enterprise Components and Process, Prentice Hall, 1999.

[11] L. Williams, R. Kessler, Pair Programming Illuminated, Addison Wesley, 2002.

[12] G. Srinivasa, P. Ganesan, "Pair Programming: Addressing Key Process Areas of the People-CMM", In Proceedings of XP/Agile Universe 2002, Chicago, August 4-7, 2002, Lecture Notes in Computer Science 2418: Springer Verlag.

[13] L. Benedicenti, R. Paranjape, "Using Extreme Programming for Knowledge Transfer", In Proceedings of XP2001 Conference, Cagliari, Villasimius, Sardinia, Italy, May 23-30, 2001.

[14] K. Schwaber, M. Beedle, R. Martin, Agile Software Development with SCRUM, Addison Wesley, 2001.

[15] A. Cockburn, Agile Software Development, Addison Wesley, 2001.

[16] S. Ambler, R. Jeffries, Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. Addison Wesley, 2002.

[17] The Wiki Web. http://c2.com/cgi/wiki

[18] S. Henninger, A. Ivaturi, K. Nuli, A. Thirunavukkaras, "Supporting Adaptable Methodologies to Meet Evolving Porject Needs", In Proccedings of XP/Agile Universe 2002, Chicago, August 4-7, 2002, Lecture Notes in Computer Science 2418:Springer Verlag.

[19] S. Mahe, C. Rieu, "Towards a Pull-Approach of KM for Improving Enterprise Flexibility Responsiveness: A Necessary First Step for Introducing Knowledge Management in Small and medium Enterprises", In Proceedings of the International Symposium on Management of Industrial and Corporate Knowledge, Compiegne, 1997.

[20] L. Prusak, M.Lesser, Communities of Practice, Social Capital, and Organizational Knowledge, IBM Institute for Knowledge Management,1999.

[21] A. Cockburn, J. Highsmith, "Agile Software Development: The People Factor", IEEE Computer, Vol. 34, No.11, 2001.

[22] P. Baheti, L. Williams, E. Gehringer, D.Stotts, "Exploring Pair Programming in Distributed Object-Oriented Team Projects", In Proceedings of XP/Agile Universe 2002, Chicago, August 4-7, 2002; Lecture Notes in Computer Science 2418: Springer Verlag, pp. 208-220

[23] D. Stotts, L.Williams, A Video-enhanced Environment for Distributed Extreme Programming. Internal Report, 2002. http://rockfish-cs.cs.unc.edu/misc/dxp-SEL02abst.pdf

[24] F. Taylor, The Principles of Scientific Management, Dover Pubns, 1998.

[25] M. Kircher, P. Jain, A. Corsaro, D. Levine, "Distributed eXtreme Programming", In Proceedings of XP2001 Conference, Cagliari, Villasimius, Sardinia, Italy, May 23-30, 2001.