Knowledge Sharing in Agile Software Teams

Thomas Chau, Frank Maurer

University of Calgary,
Department of Computer Science
Calgary, Alberta, Canada T2N 1N4
{chauth,maurer}@cpsc.ucalgary.ca

Abstract. Traditionally, software development teams follow Tayloristic approaches favoring division of labor and, hence, the use of role-based teams. Role-based teams require the transfer of knowledge from one stage of the development process to the next. As multiple stages are involved, the problem of miscommunication due to indirect and long communication path is amplified. Agile development teams address this problem by using cross-functional teams that encourages direct communication and reduces the likelihood of miscommunication. Agile approaches usually require team members to be co-located and only facilitate intra-team learning. To overcome the restriction in co-location and support organizational inter-team learning while supporting the social context critical to the sharing of tacit knowledge is the focus of this paper. We also highlight that humans are good at making sense of incomplete and approximative information.

1 Introduction

Software development is a collaborative process that needs to bring together domain expertise with technological skills and process knowledge. Traditional software development approaches organize the required knowledge sharing based on different roles following a Tayloristic mindset: people involved in the development process are assigned to specific roles (e.g. business analyst, software architect, lead designer, programmer, tester) that are associated with specific stages in the development process (requirements analysis, high-level design, low level design, coding, testing). Handoffs between each of the stages are primarily document based: one role produces a document (e.g. a requirements specification, design documents, source code, test plans) and hands it off to the people responsible for the next stage in the development process. Moving from one stage to the next often requires sign-offs from the people involved. After sign-off, documents are supposed to be set in stone (e.g. "the design is completed") and changes are seen as a problem: feature creep, requirements churn etc. Changes are handled as the exception to the rule that need to get formal approval by change control boards. Changes are also seen as a factor that dramatically (i.e. up to two orders of magnitude) increases development costs.

Tayloristic processes strive to accomplish an idealistic goal: Documents are supposed to be complete, consistent and unambiguous. Unfortunately, they never are: the information contained in the documents only approximates what is needed by the

people handling the next stage Information is lost in each transfer from one head to the next. And knowledge that does not reach the person who actually writes the code will result in a software system that does not meet all customer needs – resulting in low customer satisfaction and unsuccessful projects. A simply model illustrates this knowledge loss over longer communication chains (see Figure 1).

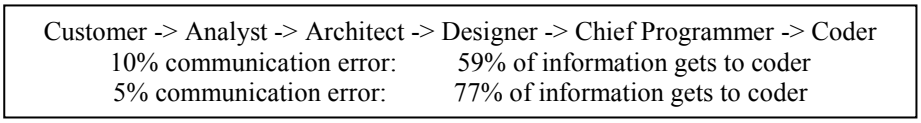

Fig. 1. Knowledge loss in Tayloristic processes

Assuming only a mere 5% of relevant information is lost in each transfer between each of the stages, nearly a quarter of the information does not reach the coder (who has to encode the domain knowledge into software) in a Tayloristic development process[1]. This gets worse if more then 5% are lost in each stage. Clearly, software engineering approaches are trying to reduce this information loss by spending effort on error correction: the main purpose of reviews and inspection is to reduce errors in all kinds of documents. While there is lots of empirical evidence showing that effort spent on inspections and reviews pays of in reduced miscommunication, nothing indicates that these processes are able to come close to a 0% error. In fact, the CHAOS report published by the Standish group shows that about three quarters of all software development projects either fail completely or are challenged. One of the main reasons for this is that the software delivered does not meet customer needs.

Another problem resulting from the long communication chains in Tayloristic software organizations is a tendency to over-document. Shannon's information theory indicates that information is only useful when it is new for the receiver of the information: providing a known fact to somebody is old news and boring. In fact, if this is done in a document, it makes the task to find relevant gems of information more difficult and, hence, increases knowledge transfer costs. People involved in the early stages of software development do not (and can not) know what information is already known to the coders. Relevance of information is completely subjective in the sense that it depends on the current knowledge of the information receiver. Based on experience, analysts and designers know that incomplete information will result in implementations that do not meet the customer's expectations. To be on the safe side and avoid problems with incomplete specifications and designs, analysts and designers tend to over-document: they answer questions with their documentation that might not even be asked by the coder.

A simple way to reduce the information loss on one hand while focusing communication on relevant information is reducing the length of the communication chain (see Figure 2).

[1] Information preserved = $(100\% - \text{error rate})^{\#\text{ of transfer}}$ (e.g. $59\% = (100\text{-}10\%)^5$)

Customer -> Developer	
10% communication error:	90% of information gets to coder
5% communication error:	95% of information gets to coder

Fig. 2. Knowledge loss in direct communication

Agile software processes like Extreme Programming [2], Scrum [4] and others [1, 3, 5, 6], rely on direct face-to-face communication between customers and developers for knowledge sharing. This reduces the information loss due to long communication chains as well as making sure that only questions that the developer (who writes the code) has are answered.

Transferring and sharing required knowledge in a team is a difficult task that, in the past, was tackled by introducing rigorous processes and more and more structured and formalized representations. While there are merits to that approach, the recent trend towards agile software processes focuses on less formal, more fuzzy style. It replaces "logical" representations by approximations - approximations that are "good enough" for humans to proceed with development but rely on face-to-face sharing of tacit knowledge to actually do so.

In this paper, we will describe how agile teams share knowledge and discuss benefits and shortcomings of these approaches. We will then present a lightweight knowledge management framework that overcomes some of the problems and explain how lightweight knowledge management can be integrated with more structured knowledge. We will also review some existing tool support for agile practices from the perspective of lightweight knowledge management.

2 Knowledge Sharing Support in Agile Processes

In agile processes, knowledge sharing is encouraged by several practices: release and iteration planning, pair programming and pair rotation, on-site customers in case of XP [2], daily Scrum meeting, cross-functional teams, and project retrospectives in Scrum [4].

Release and iteration planning are used to share knowledge on system requirements and the business domain between the on-site customers and the developers. In a release planning meeting arranged at the beginning of a project, the project timeline is broken down into small development iterations and releases. At the beginning of an iteration (short time-boxed develop efforts that run usually two to six weeks), the development team and the customer representatives discuss what should be done in the next few weeks. The discussions refine the initial requirements to a level that the development team is able to estimate the development effort for each feature. Developers break each feature into tasks and provide the customers with estimates of effort needed to complete each feature. Based on the developers' estimation, the amount of work hours available in the upcoming iteration and the velocity (the percentage spent on development task in relation to total work) from the previous iteration, prediction can be made as to whether the developers can complete the features proposed by the

customers. If not, the developers are to renegotiate the set of features with the customers. Further requirement details are discussed with on-site customer representatives while a developer actually works on the implementation of a feature. The close interaction between developers and on-site customer representatives usually lead to increased trust and a better understanding. This direct feedback loop allows a developer to create a good approximation of the requirements in his head faster than document-centric information exchange. Quickly developed software can be demonstrated immediately to the customer representative and allows her to directly catch misunderstandings.

Pair programming involves two developers working in front of a single computer designing, coding, and testing the software together. It is a very social process characterized by informal and spontaneous communications. During a pair programming session, knowledge of various kinds, some explicit but mostly tacit, is shared between the pair. This includes task-related knowledge, contextual knowledge, and social resources. Examples of task-related knowledge include system knowledge, coding convention, design practices, technology knowledge and tool usage tricks. Contextual knowledge is knowledge by which facts are interpreted and used. For instance, knowing from past experiences or "war stories" when to or when not to use a particular design pattern in different coding scenarios. Examples of social resources include personal contacts and referrals. Developers tend not to document these types of knowledge for many reasons, such as being overburdened with other tasks or they deem what they know to be irrelevant or of no interest to others. Such knowledge is often only uncovered via informal and casual conversation [7]. For this reason, the social nature of pair programming made it a great facilitator for eliciting and sharing tacit knowledge. To ensure knowledge shared among a pair is accessible to the entire team, XP recommends pairs be rotated from time to time. As a side effect of tapping tacit knowledge, the social nature of pair programming helps to create and strengthen networks of personal relationships within a team, and nurture an environment of trust, reciprocity, shared norms and values. These are critical to sustain an ongoing culture of knowledge sharing.

While pair programming sessions facilitate communication within a pair, daily Scrum meetings facilitate communication among the entire team. During a daily Scrum meeting, team members report their work progress since the last meeting; state their goals for the day; and voice problems related to their tasks or suggestions to their colleagues' tasks. Such meetings provide visibility of one's work to the rest of the team; raise everyone's awareness of who has worked on or is knowledgeable about specific parts of the system; and encourage communications among team members who may not talk to each other regularly. Team members learn whom to contact when they work on parts of the system that they are unfamiliar with.

To reduce the communication cost among the various roles, such as business analysts, developers, and testers, who are involved in software development, agile methods recommend the use of cross-functional teams instead of role-based teams. A role-based team contains only members of the same role. In contrast, a cross-functional team draws together individuals of all defined roles. Experiences indicate that cross-functional teams facilitate better collaboration and knowledge sharing which lead to reduced product development time [8].

Continuous learning is supported by some agile methods in the form of project retrospectives. Retrospectives are in essence post-mortem reviews on what happened during development except that they are conducted not only at the end of a project but also during the project. Retrospectives facilitate the identification 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 lengthy stand-up meetings, retrospectives provide the opportunity for these issues to be raised, discussed, and dealt with during the project rather than at the end of project.

3 Limitations

The above knowledge sharing practices are all team-oriented and rely on social interactions. Although the social nature of the practices made them great at tapping tacit knowledge and in fostering the creation and reinforcement of relationship networks within a team, there are inherent limitations in these practices. In their original form, all the above practices rely on face-to-face communication which restricts the use of them to co-located and small teams, usually with less than twenty people [2]. Unfortunately, for many reasons, it is sometimes impossible to co-locate an entire team and sole reliance on informal knowledge sharing will present challenges. Hence, distributed teams and inter-team knowledge sharing are issues that agile methods practitioners must deal with.

Besides the co-location constraint, the above practices only facilitate intra-team but not inter-team learning within an organization. A common attempt to address this organizational learning issue is to transfer workers from one work team to another. However, this is challenging due to cost and is slow due to time constraints.

Informal training approaches like pair programming and pair rotation are not problem-free either. Training content may vary, or conflict across different pairs. Getting two people to work cooperatively as a pair is also often an extremely tricky task. One may argue that pair programming constantly reduces the productivity of the experts as they need to train novices all the time and formal training is therefore less expensive. It should be possible to combine pair programming with a training infrastructure to gain the benefits of both approaches.

4 MASE

To overcome the above limitations, there exist various tools ranging from those that support real-time collaboration, such as Microsoft's Messenger and NetMeeting, to those that support asynchronous communication and coordination, such as e-mail and newsgroups.

While real-time collaboration tools like NetMeeting facilitate the social interaction necessary for sharing tacit knowledge, their usage is limited only to team members working at the same time. Assuming normal work hours, this is nearly impossible for teams with members working in different time zones. Likewise, tools such as e-mail and newsgroups support only asynchronous communication and collaboration.

However, the fact that "people move continually and effortlessly between different styles of collaboration: across time, across place, and so on" [9] demands tool support that can accommodate more than one collaboration style like:

- Co-located and distributed team;
- Retrieval and use of structured and unstructured information content, and;
- Synchronous and asynchronous activities.

In addition, for such a tool to be useful to agile development teams, it needs to:

- Support the social context critical to nurturing a knowledge sharing environment – providing information needs to be as easy as accessing it;
- Facilitate organizational learning, and;
- Support specific agile practices.

We will now illustrate how our proposed lightweight knowledge management platform, MASE, is able to achieve these goals and address issues stated above.

4.1 Support for Co-located and Distributed Teams

MASE is a web-based collaboration and knowledge sharing tool for agile teams. Web technology makes the tool accessible anytime anywhere by users with a web browser in their computing environment. The tool does not distinguish users working at the same place from those who work at different places. Hence, MASE is capable of supporting collaboration for both co-located and distributed teams.

4.2 Support for Unstructured and Structured Information Content

MASE allows combining the sharing of unstructured as well as structured information. Further, it makes writing (providing information) nearly as easy as reading (accessing information). Unstructured information usually consists of text and graphics. Structured information is stored in a database and, thus, must follow a schema. To support unstructured information content, the user interface of MASE is developed based on Wiki technology [10]. Wiki enable any users to access, browse, create, structure, and update any web pages in real-time using a web browser only. Each of these web pages, known as a wiki page, acts like an electronic bulletin board discussion topic with a unique name. Users use the Wiki markup language to create Wiki pages. Wiki markup is very simple (much simpler than HTML): a list of all Wiki markup commands including examples fits onto a single page.

One may argue that wiki pages are no different from any other traditional documents and will suffer from the same maintenance problems. Wiki technology mitigates this risk by automatically creating links from a wiki page to particular topics pages if the names of those topics are mentioned in that page. This helps minimize the users' effort in maintaining the relationships among the content in different wiki pages and enhance knowledge discovery. These benefits, however, are only maximized if users adhere to the same terminology when contributing content to wiki pages.

Information content in a MASE wiki page is all free-formatted text. This is not the case when users try to update a typical web page. Typically, the information content on the web page that users see is embedded among presentation information like HTML markup elements. For the users to edit such a web page, they need to spend the extra effort to first extract the information content then begin the actual editing of the content. Sometimes, this additional effort is so time-consuming that the users often give up on editing the content, thus causing knowledge content to degenerate over time. The fact that information content of web pages in MASE is in free-formatted text facilitates efficient collaboration between knowledge contributors and readers.

To support structured information content, MASE achieves this through its library of plug-ins that store specific data in a database. A MASE plug-in is usually presented as an input form that allows users to submit information or a table that displays information retrieved from a database in a structured fashion. Users can include a MASE plug-in in any wiki page simply by referencing its name. Currently, the MASE library of plug-ins includes those that are specific to agile development teams and generic team-oriented collaboration tools like rating a specific wiki page supporting collaborative filtering. The fact that any content on any wiki pages are modifiable and that any plug-ins can be included in any wiki pages give the users the flexibility to control how structured or unstructured they want their team memory to be.

Storing information does not guarantee that others can find it. And retrieving information from a repository that combines structured and unstructured data usually requires users to learn two different query mechanisms. To overcome the resulting usability problems, MASE provides full-text searching capabilities on any unstructured and structured content.

4.3 Support for Personal Portals

When a team member first logs into MASE, she is automatically provided a portal - an individual information space. She can store in her portal any content, in either structured or unstructured format. The content may be relevant only to her or to some other team members; it may not even be related to the project or task at hand. The key idea is that a team member has complete control over the type of and the granularity of the information content that she wants to see.

4.4 Support for Asynchronous and Synchronous Collaboration and Online Awareness

MASE supports asynchronous collaboration by persisting to a database the state of any wiki pages one has worked on when he/she log out of MASE. This resembles the common practice in the real world where team members leave artifacts in a physical place for others to review or update when they work at different times.

MASE also supports synchronous work through its integration with the real-time collaboration tool, Microsoft NetMeeting. Every time when a team member logs into MASE, MASE tracks the network address of that team member's computer. Through the ListUser plug-in, MASE displays which members of a team are currently using

the system, thus making all online team members aware of each other's presence. This is important for team members to establish informal and spontaneous communication with one another at ease.

4.5 Support for Agile Practices

As mentioned before, MASE supports agile practices through its library of plug-ins. For instance, project managers and customers can create iterations and user stories. MASE keeps track of all estimates made by the development team and suggests to both the development team and customers the appropriate size for the next iteration based on the developers' estimation accuracy from the previous iteration. Using the suggested iteration size, customers can prioritize user stories and move them from iteration to iteration or move them back to the product backlog. During the course of the project, both the customers and development team can track work progress at various granularities (project, iteration, user story) using the Whiteboard plug-in (see Figure 3) and view effort metrics for a particular individual or for the entire team.

Using the Whiteboard, developers can see the features and tasks allocated for each of the iterations in the project and track their time. Details of a user story or a task are stored in a wiki page allowing developers to annotate notes on them in free-formatted text. Leveraging MASE's integration with NetMeeting, developers can also perform distributed pair programming by sharing their code editor and collaborate on a design together using the shared whiteboard. Using the video and audio conferencing and multi-user text-chat features of NetMeeting, distributed team members who work at the same time can perform daily Scrum meetings.

Thus, MASE facilitates the following agile practices: release and iteration planning, distributed pair programming, collaborative design, and daily Scrum meetings.

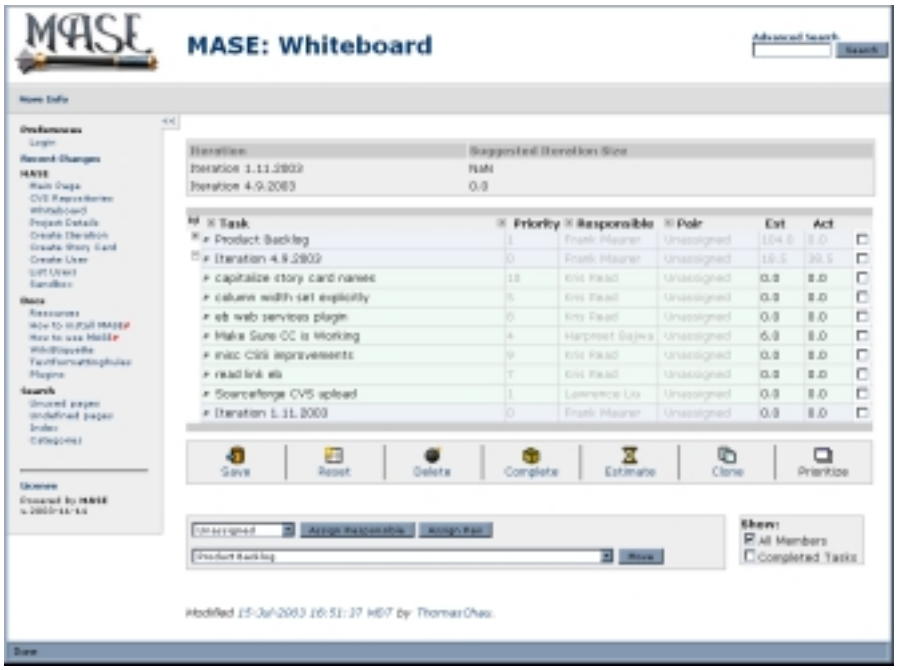

Fig. 3. MASE's project planning whiteboard

4.6 Facilitating Organizational Learning

To address the issue of organizational learning (or: inter-team learning), we adopt the view that workers learn and manage knowledge within communities of practice. We argue that facilitation for communities of practice is a critical part to support organizational learning [12]. MASE facilitates the establishment of communities of practice through its integration with the Experience Base, a tool that we have developed as an implementation of the Experience Factory concept [13] with a Wiki-like user interface similar to MASE. We illustrate MASE's support for communities of practice with the following example.

When Jill creates the task "Test shopping cart user interface" in MASE, she can associate the task with the process type "UI Testing". When Jill is ready to start working on that task, MASE will automatically create a wiki page for that task. Since the task is associated with the process type "UI Testing", Jill can embed the wiki page from the Experience Base that is dedicated to the topic "UI Testing" into the wiki page that contains details of the task she is working on. On the embedded "UI Testing" wiki page, Jill sees ideas contributed from Jack and Bill, whom she does not know and are from other teams in the company. Curious about their ideas and experiences, Jill posts her comments on the "UI Testing" wiki page.

Anecdotal evidence and thriving inter-organizational communities of practice that use Wiki servers suggest that this kind of informal knowledge sharing actually happens. By providing on-line access to the contributor of the information, MASE actually facilitates establishing direct communication Jill and Jack/Bill.

As seen from the above scenario, the integration between MASE and the Experience Base allows one to establish contact, interact, and collaborate with others who may not be working together in the same team but share common interests.

5 Related Work

Existing tools which support agile practices or inter-team learning include VersionOne [14], Xplanner [15], TWiki [16], and BORE [17]. All of them are web-based tools which allow them to be used by team members working at the same place or at different locations. However, they differ in terms of their level of support for the various agile practices, capabilities in accommodating the different collaboration styles, and facilitation for organizational learning.

Both VersionOne and Xplanner support release and iteration planning as well as project tracking. VersionOne, in particular, provides each team member with a private web page which serves as his/her own information portal. However, VersionOne predefines all the content in one's personal information portal showing only tasks that are assigned to the team member. In fact, all information content in both tools can only be created and browsed in a structured way. Team members cannot control the formality of the content nor can they specify their own search query for retrieving information.

TWiki also supports those agile team-related features provided by VersionOne and Xplanner but its usage is not targeted to agile development teams. It differentiates itself as a collaboration platform, not just a tool. This can be seen in the multitude of team-oriented tools it provides, such as event calendar, action tracker, drawing editor, and vote collection. As the name suggests, TWiki is developed based on the Wiki technology. Hence, TWiki and MASE share a lot in common: plug-in architecture, support for unstructured and structured information content, personal portal support, and full-text search. TWiki allows a set of web pages to be grouped together, known as a TWiki Web. This indirectly facilitates the establishment of communities of practice in that a community can have its own TWiki Web. One drawback of TWiki is that it provides no direct support for online team members to be aware of each other's presence. This limits the opportunities for team members to establish informal and spontaneous encounters with one another.

Unlike the other three tools, BORE does not directly support specific agile practices. It is an implementation of the Experience Factory concept. It provides an experience repository that contains experience collected from projects across the entire organization. These experiences are organized as cases, which are used to generate pre-defined tasks for a new project. A case is similar to a project task in nature. The generated set of tasks serves as a "best practice" guide. Project team members can diverge from the generated plan and not perform the suggested tasks if they deem the tasks to be inappropriate for the project situation at that time. In such cases, team members can submit their experiences and details of the tailored tasks in a structured

format to the repository. A dedicated team of people is recommended to maintain the integrity of the cases stored in the repository. Despite its explicit support for inter-team learning, BORE does not provide the supports offered by the other three tools. Its repository-centric view of knowledge sharing also does not support the social context characteristic of the knowledge sharing culture in agile teams.

6 Concluding Remarks

Traditionally, software development teams follow the Tayloristic approach favoring division of labor, hence, the use of role-based teams. Role-based teams with hand-offs between job functions have the inherent problem of amplifying the problem of miscommunication due to indirect and long communication path. Agile development teams address this problem by using cross-functional teams which encourages direct communication and reduces the likelihood of miscommunication. They rely on approximative knowledge sharing by social interaction and fast feedback loops instead of structured (logical) representations. However, there are two major inherent limitations to the various knowledge sharing practices used by agile teams in their original forms. They support only co-located teams and they do not facilitate inter-team learning. In this paper, we describe a lightweight and integrated knowledge sharing environment, MASE, which facilitates agile software development team members to

- Collaborate as co-located and distributed team;
- To engage in synchronous and asynchronous work;
- And to collaborate with members from other teams in the company via communities of practice.

MASE is available under an open-source license (http://sern.ucalgary.ca/~milos) and is used by the MASE development team as well as in undergraduate and graduate courses in several institutions.

References

1. Highsmith III, J.A. (2000), *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems,* Dorset House Publishing.
2. Beck, K (2000), *Extreme Programming Explained: Embrace Change,* Addison Wesley, Reading, MA.
3. Stapleton, J. (1997), *DSDM Dynamics System Development Method,* Addison Wesley, Reading, MA.
4. Beedle, M., Schwaber, K. (2001), *Agile Software Development with SCRUM,* Prentice Hall, Englewood Cliffs, NJ.
5. Cockburn, A. (2002), *Agile Software Development*; Addison Wesley, Reading, MA.
6. Ambler, S., Jeffries, R. (2002), *Agile Modeling: Effective Practice for Extreme Programming and the Unified Process*, Addison Wesley, Reading, MA.

7. Fitzpatrick G. (2001), Emergent Expertise Sharing in a New Community, in M.S. Ackerman, P. Volkmar & W. Volker, eds, *'Sharing Expertise: Beyond Knowledge Management'*, MIT Press, Cambridge, MA.
8. Haas, R., Aulbur, W., Thakar, S. (2000), Enabling Communities of Practice at EADS Airbus, in M.S. Ackerman, P. Volkmar & W. Volker, eds, *'Sharing Expertise: Beyond Knowledge Management'*, MIT Press, Cambridge, MA.
9. Greenburg, S., Roseman, M. (1998), Using a Room Metaphor to Ease Transitions in Groupware, in M.S. Ackerman, P. Volkmar & W. Volker, eds, *'Sharing Expertise: Beyond Knowledge Management'*, MIT Press, Cambridge, MA.
10. Cunningham, W., Leuf, B. (2001), *The Wiki Way Quick Collaboration on the Web,* Addison Wesley, Reading, MA.
11. JSPWiki http://www.jspwiki.org (Last Visited: September 25, 2003)
12. Erickson, T., Kellogg, W. (2001), Knowledge Communities: Online Environments for Supporting Knowledge Management and Its Social Context, in M.S. Ackerman, P. Volkmar & W. Volker, eds, *'Sharing Expertise: Beyond Knowledge Management'*, MIT Press, Cambridge, MA.
13. Basili, V., Caldiera, G., Romback, H. (1994), "Experience Factory", In *Encyclopedia of Software Engineering vol. 1,* J.J. Marciniak, Ed. John Wiley Sons.
14. VersionOne http://www.versionone.net (Last Visited: September 25, 2003)
15. Xplanner http://www.xplanner.org (Last Visited: September 25, 2003)
16. TWiki http://www.twiki.org (Last Visited: September 25, 2003)
17. Henninger, S., Ivaturi, A., Nuli, K., Thirunavukkaras, A. (2002), "Supporting Adaptable Methodologies to Meet Evolving Project Needs", in D. Wells, L. Williams, eds, *Proceedings of XP/Agile Universe 2002,* Springer, Berlin Heidelberg New York.