Knowledge Management in Agile Projects

Executive Summary

Software development is knowledge-intensive work and the main challenge is how to manage this knowledge. The Agile manifesto advocates "individuals and interaction over process and tools," and hence it requires even more attention to manage knowledge in Agile projects.

This paper demarcates the types of knowledge involved in the lifecycle of software projects and describes the mechanisms to effectively manage them in Agile software development. It then argues for the need to scale Agile development strategies in knowledge management to address the full delivery process.

Knowledge Management in Software Development

Knowledge management is "a method that simplifies the process of sharing, distributing, creating, capturing and understanding the company knowledge."[1] Knowledge itself "is a fluid mix of framed experience, values, contextual information and expert insight that provide a framework for evaluation and incorporating new experience and new information."[2] Furthermore, "knowledge passes through different modes of conversion, which makes the knowledge more refined and spreads it across different layers in an organization."[3]

The main assets of software development are not manufacturing plants, buildings and machines but the knowledge held by the employees and the development culture of an organization. Companies developing information systems have failed to learn effective means for problem solving to such an extent that they have learned to fail. The key drivers for companies to manage knowledge effectively in software development are:

- Reducing the effort spent in acquiring required knowledge for project execution.
- Improving reusability (i.e., avoiding reinvention).
- Reducing dependency on individuals for project success.
- Improving the overall team's productivity.

Knowledge Types

Typically, knowledge can be classified into two types, explicit and tacit.

Explicit knowledge is knowledge that is articulable and transmittable in formal, systematic language. This can include grammatical statements, mathematical expressions, specifications, manuals and so forth. Such knowledge can be transmitted formally among individuals with ease.

Tacit knowledge is personal and context-specific, and is therefore difficult to formalize and communicate. It is embedded in individual experience and involves intangible factors such as personal belief, perspectives and value systems. Tacit knowledge

Cognizant

is difficult to communicate and share in an organization and thus must be converted into words or other forms of explicit knowledge.

The Knowledge Lifecycle in Software Projects

The knowledge life cycle in software projects can be described in five steps.

1. **Gather available explicit knowledge.** Generally, this happens during the start of the project whereby available knowledge is captured and made available in knowledge bases, marketing, different departments, etc.
2. **Personalize explicit knowledge.** The gathered information/knowledge needs to be converted by the individuals involved in the projects into tacit knowledge.
3. **Application of the acquired knowledge.** Converted tacit knowledge will be applied to the project execution.
4. **Learning from the project.** There could be some learning, innovations or new methods or techniques uncovered during the execution of the project which will be in the form of tacit knowledge.
5. **Convert to explicit knowledge.** The learning needs to be converted into explicit knowledge and added to the repository for future reference.

There are many questions related to knowledge management, especially in Agile projects: How different is it to manage knowledge in Agile projects? What should be the relative levels of focus on explicit knowledge vs. tacit knowledge? This paper will address these queries.

The Knowledge Circle

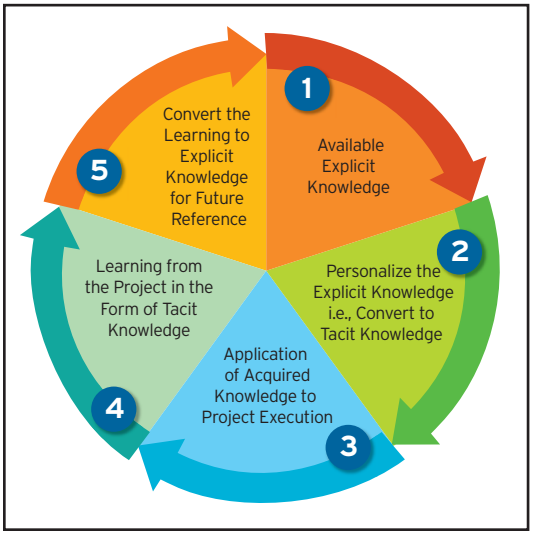

Figure 1

Knowledge Management in Traditional Software Development

Traditional software development approaches organize the required knowledge sharing based on different roles following a Tayloristic[4] mind-set: people involved in the development process are assigned specific roles (e.g., business analyst, software architect, lead designer, programmer, tester, etc.) that are associated with specific stages in the development process (requirements analysis, high-level design, low-level design, coding, testing, etc.).

Limitations

Knowledge sharing between each of the stages is primarily document based (i.e., through explicit knowledge). One role produces a document (e.g., requirements specifications, design documents, source code, test plans, etc.) and hands it off to the people responsible for the next stage in the development process.

Assuming merely 5% of relevant information is lost in each transfer between 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. The results are even worse if more than 5% is lost in each stage.

Another problem resulting from the long communication chains in Tayloristic software organizations is a tendency to over-document. Information is useful only when it is new to the receiver; providing a known fact to somebody is old, boring news. In fact, such repetition makes it more difficult to find the relevant "gems" of information in a document and hence increases knowledge transfer costs. People involved in the early stages of software development do not (and cannot) 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.

Knowledge Management in Agile Software Development

Agile software development relies on direct communication – i.e., synchronized and osmotic communications between customers and developers for knowledge sharing. This reduces the information loss due to long communication chains and

it ensures 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 traditional model 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 a less formal, "fuzzier" style. It replaces "logical" representations by approximations – approximations that are "good enough" for humans to proceed with development but rely on the sharing of tacit knowledge to actually do so.

In Agile processes, knowledge sharing is encouraged by several practices:

- Release and iteration planning.
- Pair programming and pair rotation.
- Daily Scrum meetings.
- Cross-functional teams.
- Retrospectives.

Release and iteration planning are used to share knowledge on system requirements and the business domain between on-site customers and 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 (a short, time-boxed development effort that runs 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. 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 leads to increased trust and better understanding. This direct feedback loop allows a developer to express a good approximation of the requirements in his head faster than document-centric information exchange could. 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" whether or 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 deeming what they know to be irrelevant or of no interest to others. Such knowledge is often only uncovered via informal and casual conversation.

The social nature of pair programming make it a great facilitator for eliciting and sharing tacit knowledge. To ensure knowledge shared among a pair is accessible to the entire team, it is recommended to rotate pairs 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 to 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/suggestions related to their tasks or 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 that are involved in software development — such as business analysts, developers and testers — Agile methods recommend the use of cross-functional teams instead of role-based teams. A role-based team contains only members in the same role. By 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 leads to reduced product development time.

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 it. 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 the project.

Limitations

Although the concept of learning is embedded in various Agile software development practices, as shown above, these practices only address knowledge sharing within a team. They do not address issues of knowledge sharing across team boundaries. In a large organization, there may exist multiple teams that work on similar tasks, face common problems or have overlapping interests in specific knowledge areas. In short, there is a lack of explicit support for organizational learning. Also, there will be instances where Agile teams are distributed due to the nature of business, which also makes tacit knowledge sharing difficult.

To address this, we need to have lightweight mechanisms to convert tacit knowledge into explicit knowledge in Agile software development. Some of the mechanisms are as follows.

Agile teams should make it a practice to capture learns in a particular Sprint as part of every retrospective meeting and ensure that this is converted into explicit knowledge (through documentation) and published in a common area that everyone can access. The common area could be a lightweight collaboration Website like a Wiki which should act as a collaborative knowledge repository for the team.

The majority of the information captured in traditional specification documents, such as requirements specifications, architecture specifications or design specifications, can be captured as "executable specifications" in the form of tests. When you take a test-driven development (TDD) approach, you effectively write detailed specifications on a just-in-time (JIT) basis. With TDD, you write a test, either at the customer/acceptance level or the developer level, before writing sufficient code/functionality to fulfill that test. The tests accomplish two purposes: they specify the requirements/architecture/design and they validate your work. This is an example of the practice of single source information.

Make it a practice to create needy documents — i.e., if and only if they fulfill a clear, important, and immediate goal of overall project efforts. Don't forget that this purpose may be short-term or long-term; it may directly support software development efforts or it may not. Also, remember that each system has its own unique documentation needs, that one size does not fit all. The implication is that you're not going to be able to follow a "repeatable process" and leverage the same set of documentation templates on every project, at least not if you're interested in actually being effective.

Conclusion

Traditionally, software development teams follow the Tayloristic approach favoring division of labor; hence, the use of role-based teams. Role-based teams with handoffs between job functions have the inherent problem of amplifying the problem of miscommunication due to indirect and long communication paths (i.e., knowledge sharing through explicit knowledge).

Agile development teams address this problem by using cross-functional teams, which encourages direct communication through release and iteration planning, pair programming and pair rotation, and daily stand-ups and retrospectives — all of which reduces the likelihood of miscommunication. They rely on various practices which emphasize approximate knowledge sharing by

social interaction and fast feedback loops instead of structured (logical) representations (i.e., knowledge sharing through tacit knowledge).

However, there are major inherent limitations to the various knowledge-sharing practices used by Agile teams in their original forms. They do not facilitate inter-team learning and also do not work well if the teams are distributed. To address this, Agile teams should not only rely on the knowledge sharing through tacit knowledge but should also employ lightweight explicit knowledge sharing like test-driven development, needy documentation, usage of Wikis, and the discipline of capturing the learns through retrospectives.

Footnotes

[1] T.H.Davenprot, L.Prusak, "Working Knowledge: How Organizations Manage What They Know," Harvard Business School Press, Boston, 1998.

[2] Argyris C., "Knowledge for Action," (1993), San Francisco, CA: Jossey-Bass.

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

[4] Frederick W. Taylor was an American engineer who introduced scientific factory management in the early 19th century. His innovations in time and motion studies paid off in dramatic improvements in productivity which favors division of labor and, hence, the use of role-based teams.

About the Author

Vadivelan Sivanantham is a Senior Manager in Cognizant's Advanced Solutions Group. He is a full-time Agile Scrum Coach and has CSM certification from the Scrum Alliance. Vadivelan specializes in performing Agile assessments and coaching Waterfall to Agile transitions. His software experience includes many years of handling projects in both Agile and traditional methodologies. Vadivelan received a bachelor's degree in engineering from Anna University located in Chennai, India and a master's degree in information technology from Bharathidasan University located in Trichy, India. He can be reached at Vadivelan.Sivanantham@cognizant.com.

About Cognizant

Cognizant (NASDAQ: CTSH) is a leading provider of information technology, consulting, and business process outsourcing services, dedicated to helping the world's leading companies build stronger businesses. Headquartered in Teaneck, New Jersey (U.S.), Cognizant combines a passion for client satisfaction, technology innovation, deep industry and business process expertise, and a global, collaborative workforce that embodies the future of work. With over 50 delivery centers worldwide and approximately 130,000 employees as of September 30, 2011, Cognizant is a member of the NASDAQ-100, the S&P 500, the Forbes Global 2000, and the Fortune 500 and is ranked among the top performing and fastest growing companies in the world. Visit us online at www.cognizant.com or follow us on Twitter: Cognizant.

Cognizant

World Headquarters
500 Frank W. Burr Blvd.
Teaneck, NJ 07666 USA
Phone: +1 201 801 0233
Fax: +1 201 801 0243
Toll Free: +1 888 937 3277
Email: inquiry@cognizant.com

European Headquarters
1 Kingdom Street
Paddington Central
London W2 6BD
Phone: +44 (0) 20 7297 7600
Fax: +44 (0) 20 7121 0102
Email: infouk@cognizant.com

India Operations Headquarters
#5/535, Old Mahabalipuram Road
Okkiyam Pettai, Thoraipakkam
Chennai, 600 096 India
Phone: +91 (0) 44 4209 6000
Fax: +91 (0) 44 4209 6060
Email: inquiryindia@cognizant.com