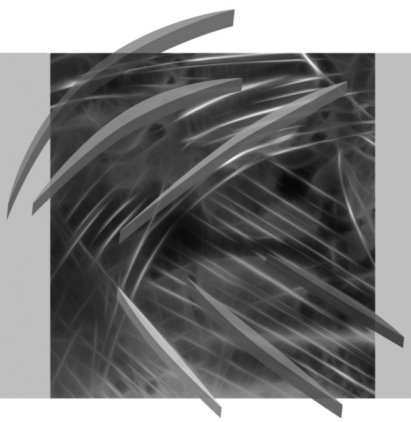
PRACE NAUKOWE
Uniwersytetu Ekonomicznego we Wrocławiu
RESEARCH PAPERS
of Wrocław University of Economics

232

Knowledge Acquisition and Management



edited by
Małgorzata Nycz
Mieczysław Lech Owoc

Publishing House of Wrocław University of Economics
Wrocław 2011

Reviewers: Grzegorz Bartoszewicz, Witold Chmielarz, Halina Kwaśnicka,
Antoni Ligęza, Stanisław Stanek

Copy-editing: Marcin Orszulak

Layout: Barbara Łopusiewicz

Proof-reading: Barbara Łopusiewicz

Typesetting: Beata Mazur

Cover design: Beata Dębska

This publication is available at www.ibuk.pl

Abstracts of published papers are available in the international database
The Central European Journal of Social Sciences and Humanities
http://cejsh.icm.edu.pl and in The Central and Eastern European Online Library
www.ceeol.com as well as in the annotated bibliography of economic issues BazEkon
http://kangur.uek.krakow.pl/bazy_ae/bazekon/nowy/index.php

Information on submitting and reviewing papers is available
on the Publishing House's website www.wydawnictwo.ue.wroc.pl

ISSN 1899-3192
ISBN 978-83-7695-200-0

The original version: printed

Printing: Printing House TOTEM

Contents

Presentations

Streszczenia

PRACE NAUKOWE UNIWERSYTETU EKONOMICZNEGO WE WROCŁAWIU nr 232
RESEARCH PAPERS OF WROCŁAW UNIVERSITY OF ECONOMICS
Knowledge Acquisition and Management ISSN 1899-3192

Radosław Zatoka, Cezary Hołub
Wrocław University of Economics

KNOWLEDGE MANAGEMENT IN PROGRAMMING TEAMS USING AGILE METHODOLOGIES

Summary: This paper discusses the idea of knowledge management in software development organisations at the level of programming teams. The authors use conceptual modelling approach to adapt classical knowledge flow cycle for teams of programmers building projects by means of agile development processes. In our work we select techniques derived from agile methodologies and artefacts of software development process and integrate them into Nonaka and Takeuchi's concept in order to obtain a more coherent model – the proposed extension of Nonaka and Takeuchi's model conforms to the discipline of software engineering and the character of agile projects.

Keywords: knowledge management, agile development methodologies, software engineering, programming teams.

1. Introduction

Software development is a highly heterogonous process, involving many disciplines, among which one can distinguish requirements engineering, systems analysis and design, software engineering, user interface design, graphic design, project management, etc. As a consequence, this field requires knowledge intensively in various areas. Fastly changing and steadily growing knowledge needs to force software development companies to introduce knowledge management strategies which can facilitate tasks that concern capturing and using knowledge. This paper focuses on knowledge management in programming teams using agile methodologies. The software development context is narrowed down to implemental, code-creational level. Knowledge management in software engineering, and in particular, from the perspective of a member of a programming team, can be distanced from the common knowledge management, but the classic knowledge management concepts can be used as a starting point.

2. Method

A systematic review and a conceptual modelling approach was used in this paper. The authors start with referring to the knowledge management classical theory and distinguish explicit and tacit knowledge assets in software development. We continue

our deliberation by setting knowledge management beside agile development methodologies and highlighting the role of tacit assets in agile teams. The theoretical part is finished with the short comparison between sequential design process and agile methodologies. The last part of the paper presents modified Nonaka and Takeuchi's model adapted to the character of agile projects. This section emphasises distinguishing and discussing technics and methods derived from agile methodologies that support creation of tacit knowledge assets. It also reviews a need for their codification to maintain a project in the long run.

3. Knowledge in software development organisations

As presented in Nonaka and Takeuchi's model, intellectual capital of an organisation appears in the form of tacit and explicit knowledge [Nonaka, Takeuchi 1995]. Tacit knowledge is personal knowledge of employees which they bring to a company and continuously develop while "learning-by-doing". Explicit knowledge is codified; it can be represented in textual or symbolic form and easily shared in a group. One can say that from the perspective of a programmer tacit knowledge reflects changes in practice, while explicit knowledge creates an organisation memory.

The endeavour of application Nonaka and Takeuchi's distinction at the programming team level can result in distinguishing tacit and explicit knowledge assets. Table 1 presents the most important assets.

Table 1. Explicit and tacit knowledge assets in the process of software development

Explicit assets	Tacit assets
• Documented code repositories • System specifications and client requirements • System fragments models • Unit-tests sets • Reports from team meetings	• Skills and experience of team members • Undocumented code repositories

Source: authors' own study.

The general idea of knowledge management systems emphasises transforming tacit knowledge into explicit. The proportion of managing these two knowledge asset groups is connected with used software development method. Teams using agile development processes should concentrate on tacit assets, while the explicit ones are more important when using traditional sequential design methods, for instance waterfall. However, according to the research in the field of software engineering, both explicit and tacit knowledge are required, no matter which approach is pursued. [Bjørnson, Dingsøyr 2008].

4. "Programming as Theory Building"

In 1985 Peter Naur published his article "Programming as Theory Building", in which he introduced the *Theory Building View* of programming [Naur 1992]. The idea, however, became more known and widely commented just not long ago, with the works of Alistair Cockburn, the co-author of the Agile Manifesto.

Naur's concept derived from observations that "at least with certain kinds of large programs, the continued adaptation, modification, and correction of errors in them, is essentially dependent on a certain kind of knowledge possessed by a group of programmers who are closely and continuously connected with them" [Naur 1992]. This leads to a conclusion that essential part in programming is building up programmers' knowledge, regarded as a theory which is intangible and created throughout the process of software development. Without having the proper theory, a person is not able to do certain things intelligently. Coping with changes in software development demands matching various theories: the theory of a client problem, the theory of programmers solution, and often theories of previous programmers.

Theory Building View points out that "knowledge of the theory is tacit in the owning" [Cockburn 2008]. An important consequence of this idea is that restoring the theory of a programme merely from the documentation is strictly impossible. A programme revival depends upon transmitting the theory which requires passing on both explicit and tacit knowledge. In next sections we briefly describe and compare waterfall model to agile methodology to present the difference in the way they impact knowledge assets in a programming team.

5. Software process methods

"Software process" is described as a process of producing software. Software has been produced for a relatively short time, because the manufacturing processes of software change quickly over time, often changing the opinion on "who is the best". There are a lot of schools talking about how to make software – they provide the most variations of the two major (waterfall and agile) – and extremely different in relation to each process:

– *Rational Unified Process* (called the RUP): the process of software developed by Rational Software; the process is adapted to carry out major and very big projects;
– *Agile* (Scrum): the software is created in the specified intervals, so-called "sprints", during which a team is to perform pre-defined job requirements. Formalisation is low.

In Figure 1 there is an example project structure regarding waterfall methodologies. We can see that there are three levels (groups) of knowledge: project manage-

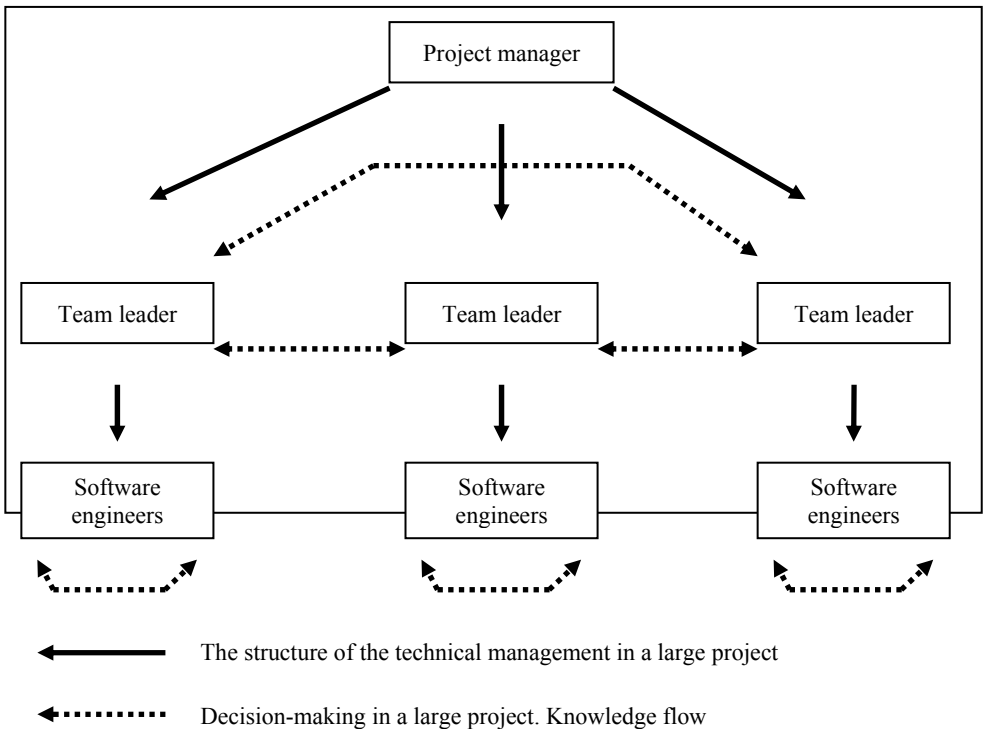


Figure 1. The structure of the technical management, decision-making, and knowledge flow in a large project

Source: authors' own study.

ment, team leaders, and software engineers. People at each level cooperate with one another but not between levels. This is not a really good approach. Agile methodologies try to find a panacea for such a situation.

6. Comparison of agile and waterfall methodologies

In the case of waterfall methodology, agile methodology and the difference in the approach to the process of software development is significant. Cascade model is dedicated to a work plan (called a plan-driven), which means that at the outset of a project we know exactly the scope of our work; it only affects time and cost. In the case of agile, methodology is the opposite, having a predetermined budget and time frame, so manoeuvring the scope of work to best meet the needs of a client (called value/vision-driven), as illustrated in Figure 2. One of the key issues was client involvement in a project so that it becomes an integral part of a team. Thus, we have a chance to create software which 100% satisfies customer expectations.

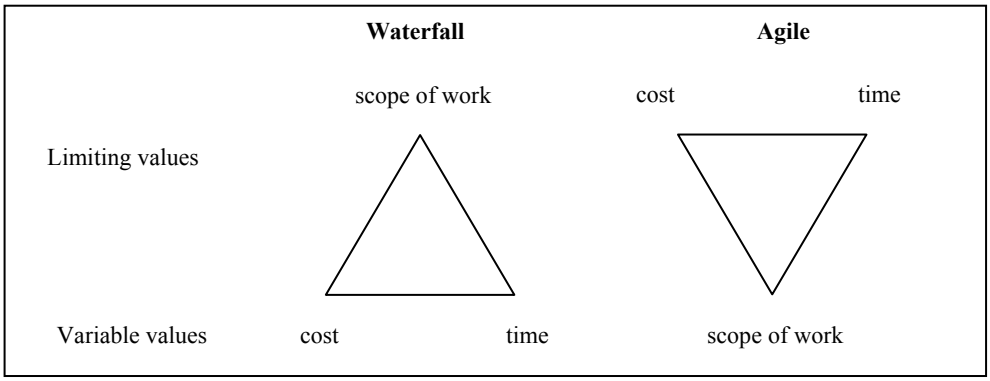


Figure 2. The differences between agile methodology and the waterfall

Source: authors' own study.

Another distinct feature is the division of work. In the waterfall methodology the tasks are completed in a cascade. We collect customer requirements, make their analysis, and on this basis we can design the whole program; therefore, coding, testing, and finally commissioning a project to a client. A negative side of the model is that the accelerated coding phase threatens the pace of work at the end (nervous atmosphere and time pressure). Deadline is often at the expense of testing the effect of what a customer gets is not always fully tested programme. Agile as a methodology to promote the TDD technique (called Test Driven Development), does not leave testing to the very end of work. A kind of innovation is that tests are first prepared before developers proceed to a proper coding. The whole project is divided into various iterations and for each of these steps a separate testing unit is provided.

7. Knowledge flow in programming teams

Among factors which influence the success of software development projects researchers list: new technologies, methods of software engineering, and people performance [Rus, Lindvall 2002]. To improve the quality and lower the costs of software development, knowledge management systems affect the latter.

Due to Nonaka and Takeuchi [1995], knowledge is converted from tacit to explicit and *vice versa* as it flows through an organisation in the processes of socialisation, externalisation, internalisation, and combination. In the software development context, Nonaka and Takeuchi's model can be adapted and extended to present knowledge cycle in the programming team. It is shown in Figure 3.

As stated earlier, agile software engineering is mainly tacit-knowledge oriented. Therefore, managing human factors is the key to a success in creating efficient knowledge managements systems at the level of a programming team using agile

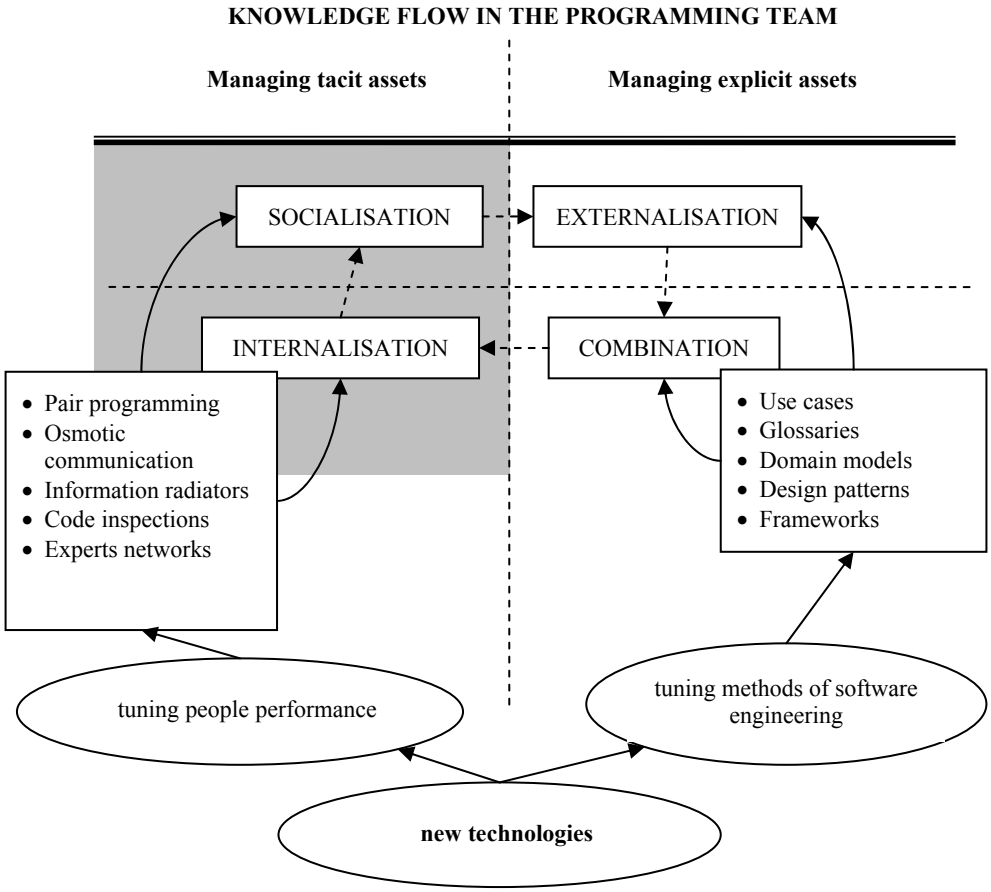


Figure 3. Knowledge flow in a programming team

Source: authors' own study.

methodologies. Explicit assets are not directly relevant and have an auxiliary role in this process.

Among methods and techniques deriving from agile methodologies, we suggest using those listed further in the process of creating and managing programming teams. These methods of transforming tacit assets into tacit knowledge involve creating:

– **experts networks** in a company that try to indicate who knows what, connect programmers with particular knowledge and encourage them to exchange views even if they currently work on different projects [Schneider 2009];
– **osmotic communication** in a programming team, which is ensured a shared office, where programmers can interact and find seating arrangements, where one or more business experts can sit close to two or more programmers; osmotic

communication will significantly lower the cost of ideas transfer [Cockburn 2008];
- **information radiators** that display information in a freely available place, where passing team members can be exposed to their effects [Cockburn 2008];
- **code inspection phase**, in which members of an inspection team can revise the code written by their colleagues, inform them about their mistakes and lapses, and give them an instant feedback [Martin, Micah 2008];
- **pair programming:** extreme programming or side-by-side programming.

Pair programming is a technique introduced by agile methods in which two programmers work on the same task simultaneously. In extreme programming variant, programmers are equipped with one computer. While the first one is writing a code, the second gives instant advice, reviews the code, and asks questions. Extreme programming allows transferring information in the real time. The side-by-side variant assumes that programmers have their own computers, but the pair works close to each other, consulting the whole task and dividing it together into subtasks.

According to the research conducted at Poznań University of Technology [Nawrocki et al.], however, the effort overhead for side-by-side programming is 20% and for XP programming about 50%, in comparison to the classic style. Moreover, the verification phase of the experiment did not confirm that persons programming in pairs had better code familiarity than individuals. In the light of those results, pair programming may seem as a controversial and risky technique.

Although agile development processes concentrate on tacit assets, explicit assets are required for further maintenance of a project. To support the process of managing explicit assets, software engineering developed many structures that enclose knowledge in reusable, codified forms. They include [Schneider 2009]:
- **use cases**, sets of functional requirements that occur in an interaction between users and a computer system;
- **glossaries**, dictionaries of terminology belonging to the modelled domain;
- **domain models** that capture key concepts of the domain logic and present how objects distinguished within are related;
- **design patterns**, programmers experience, codified as a combination of a problem, solution, and context; software development patterns can be applied at both architectural and implementation levels.

These artefacts of software process can be used to transform tacit knowledge into explicit and, by placing them in document repositories, lift knowledge assets from the individual to the organisational level.

8. Conclusions

Agile methods, which are still growing in popularity, introduce a number of innovative solutions, such as pair programming or parallel programming, which can improve the knowledge cycle in a programming team. Even if these methods are not used in a project as a whole, project managers should consider their partial utilisation.

Knowledge management system for teams of developers should concentrate on managing human factor: encouraging the integration of individuals and creating the environment for sharing ideas and skills. Managing explicit assets can be facilitated by a subsequent use of the artefacts, which represent the theory of the system – a codification of knowledge, created at all the stages of software development process. Further research may focus on empirical model verification tests.

It is worth noting that a threat to knowledge management systems in software development industry can be programmers themsleves. Project managers should not only motivate employees to share knowledge but also reward knowledge by keeping key people in an organisation. On the contrary, it cannot lead to a situation in which a programmer feels confident that he or she is in the possession of a unique knowledge and a company is dependent on him or her. We must promote an approach stating that sharing knowledge does not carry risks but profits.

References

Bjørnson F.O. (2007), *Knowledge Management in Software Process Improvement*, Doctoral Thesis, Norwegian University of Science and Technology.

Bjørnson F.O., Dingsøyr T. (2008), Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used, *Information and Software Technology*, Vol. 50.

Cockburn A. (2008), *Agile Software Development. Gra zespołowa*, Wydanie II, Helion.

Davenport T.H. (2010), *Knowledge Management Case Study, Knowledge Management at Microsoft*, http://www.itmweb.com/essay536.htm (date of access: 29.04.2010).

Lindvall M., Rus I., Jammalamadaka R., Thakker R. (2001), *Software Tools for Knowledge Management. A DACS State-of-the-Art Report*, The University of Maryland.

Liu Ch.-H., Wong R., Chen Y.-T., Huang H.-W. (2006), Managing critical knowledge management issues in global software development project, *Issues in Information Systems*, Vol. VII, No. 2.

Martin R.C., Micah M. (2008), *Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#*, Helion.

Merta A. (2008), Była sobie inspekcja. Określenie, przygotowanie i definicja procesu inspekcji, *Software Developer's Journal*, nr 9.

Naur P. (1992), *Programming as Theory Building, Computing: A Human Activity*, ACM Press, pp. 37-48.

Nawrocki J.R., Jasiński M., Olek Ł., Lange B. (2005), *Pair Programming vs. Side-by-Side Programming*, EuroSPI 2005, Budapest, November 2005.

Nonaka and Takeuchi Knowledge Management Cycle (2010), http://hubpages.com/hub/Nonaka-and-Takeuchi-knowledge-management-cycle (date of access: 29.04.2010).

Nonaka I., Takeuchi H. (1995), *The Knowledge-Creating Company*, Oxford University Press, New York.

Object Manegement Group (OMG), http://www.omg.org.

Rus I., Lindvall M. (2002), *Knowledge Management in Software Engineering*, Fraunhofer Center for Experimental Software Engineering, Maryland.

Schneider K. (2009), Knowledge Management in Software Engineering, Springer-Verlag, Berlin/Heidelberg.

Shore J., Warden S. (2008), *Agile Development. Filozofia programowania zwinnego*, Helion.
Pair Programming vs. Side-by-Side Programming, http://xprince.net/xprince_folder/artyku142y (date of access: 29.04.2010).

ZARZĄDZANIE WIEDZĄ W ZESPOŁACH PROGRAMISTYCZNYCH PRZY UŻYCIU METODYK ZWINNYCH

Streszczenie: Artykuł omawia ideę zarządzania wiedzą w organizacjach tworzących oprogramowanie na poziomie zespołów programistycznych. Autorzy używają podejścia modelowania konceptualnego, aby zaadaptować klasyczny przepływ wiedzy na potrzeby zespołów programistycznych realizujących projekty przy użyciu metodyk zwinnych. W naszej pracy wybieramy techniki pochodzące z metodyk zwinnych oraz artefakty procesu rozwoju oprogramowania i integrujemy je z koncepcją Nonaki i Takeuchiego w celu uzyskania bardziej spójnego modelu. Zaproponowane rozszerzenie modelu Nonaki i Takeuchiego dostosowuje się do dyscypliny związanej z rozwojem oprogramowania oraz do charakteru projektów realizowanych w inżynierii oprogramowania oraz w metodykach zwinnych.

Słowa kluczowe: zarządzanie wiedzą, zwinne metodyki rozwoju oprogramowania, inżynieria oprogramowania, zespoły programistyczne.