An Architecture to Support Learning, Awareness, and Transparency in Social Software Engineering

doi:10.3991/ijet.v5s1.1194

Wolfgang Reinhardt and Sascha Rinne
University of Paderborn, Paderborn, Germany

***Abstract*—Classical tools for supporting software engineering teams (collaborative development environment, CDE) are designed to support one team during the development of a product. Often the required data sources or experts reside outside of the internal project team and thus not provided by these CDEs. This paper describes an approach for a community-embedded CDE (CCDE), which is capable of handling multiple projects of several organizations, providing inter-project knowledge sharing and developer awareness. The presented approach uses the mashup pattern to integrate multiple data sources in order to provide software teams with an exactingly development environment.**

***Index Terms*—Learning Systems, Knowledge Management, Cooperative Development Environments, Learning Communities**

I. INTRODUCTION

Traditional clichés about software developers loose their validity more and more. Times, when programmers sat in dark cellars and tried to solve all problems on their own are over once and for all. In the meantime software engineering has become a very knowledge-intensive [5] and communicative process (not only but also triggered by agile methods for software development) where the actors heavily exchange data (see Google-Code[1]), connect with like-minded (see Google Summer of Code[2]), blog about experiences in their own weblogs, provide code snippets free of charge (see Django-Snippets[3]) or help novices with words and deeds in large mailing lists. This *social software engineering* – the creation of software and related artefacts within a social network – gained a lot of attention in recent software engineering research [1,17]. Besides the improvements of integrated development environments (IDE, e. g. Eclipse [4]) or procedure models (e.g. eXtreme Programming [3]) research is addressing improvements of the daily working and learning environments of the developers. The main function of collaborative development environments (CDE) [2] is to support the whole development process of a team of software developers from start to finish. This includes version control of code artefacts as well as process documentation, coordination of tasks or support for division of labour.

CDEs usually are set up for one specific project; the possibilities for inter-project-collaboration within an organization with multiple software projects are very limited because the single CDEs are not able to exchange data.

[1] http://code.google.com/
[2] http://code.google.com/soc/
[3] http://www.djangosnippets.org/
[4] http://www.eclipse.org/

Furthermore many developers are using data pools (bulletin boards, developer communities, mailing lists and a lot more) outside the organization in order to solve a specific problem. Furthermore existing CDEs lack in providing a transparent view on the progress of a project, awareness of developers' competencies and support for individual informal learning processes.

This paper describes an approach for a *community-embedded CDE* (CCDE), which is capable of handling multiple projects of several organizations, providing inter-project knowledge sharing and developer awareness. The presented approach uses the mashup pattern to integrate multiple data sources in order to provide software teams with an exactingly development environment. Furthermore we present requirements for a community of developers and sketch a first prototypical architecture for such a CCDE.

II. RELATED WORK

The goal of this section is to behold the main aspects enlisted in the conception and implementation of a CCDE in order to derivate functional and technical requirements. Furthermore this section serves for definition and dissociation of the used terms.

A. *Knowledge Management and Learning in Software Engineering*

The different facets of the concept of knowledge have been discussed for over 2000 years now. Based on a fuzzy understanding of knowledge several theories for knowledge management came up and raised the idea of simply exchanging knowledge between individuals or organizations (among others [8]). It is probably the most important assessment to be made in this context that *„you cannot store knowledge“* [7] as in interpersonal communication only data is exchanged. Information emerges by interpreting this data with own prior knowledge in the personal context. Information then is the foundation for personal actions and decisions. So knowledge is first of all a rational capacity and not a transferable item. POLYANI distinguishes between tacit and explicit knowledge, whereas explicit knowledge is stored in textbooks, software products and documents, while tacit knowledge is in the mind of people as memory, skills, experience and creativity [10]. When tacit knowledge is externalised and transformed into explicit knowledge (properly speaking it is data now), we call this implicit knowledge. Implicit knowledge is of very high value for organisations such as software projects, as it gives hints how to solve specific problems in the future.

Regardless of the ambiguous definitions of knowledge and the claims for necessity and importance for knowledge management, software engineering is a dynamic process, which is reliant on latest knowledge in the subject domain. This knowledge is dynamic and evolves with technology, organisational culture and changing needs of the organisation [9]. Knowledge management in software engineering can be improved by recognising the need for informal communication and exchange of data in order to support the exchange of implicit knowledge amongst developers. Learning and working environments thus should support awareness of developers, sharing of implicit knowledge and foster informal, ad hoc exchange of short messages [6,11] as well as facilitating inter-project social networks in form of communities of interest.

Informal learning is characterized as a process that does not follow a specified curriculum but rather happens by accident, sporadically and naturally during daily interactions and shared relationships. Experience shows that the majority of real learning is informal [4]. Informal learning is what happens when tacit knowledge of a person is communicated to another person, which internalizes and interprets the data and thus expands his own knowledge. Examples of such informal learning situations with in software engineering projects are spontaneous meetings, short messages, phone calls but also asynchronous communication like entries in bulletin boards, comments in source code or comments in blogs. As hardly any formal training for developers takes place, in software engineering informal learning is the only way to stay up to date. Previous approaches for supporting ad hoc communication focus on intra-project improvements and do not include experts from outside the project. Connecting with others and using artefacts from outside the own project seem to be a crucial factor in supporting learning within a project.

B. *Social Software Engineering*

The term social software engineering denotes both the engineering process of so called social software and the software engineering within social relationship in collaborative teams. For this paper the latter denotation is the focus of interest.

Studies show, that the main part of modern software engineering is carried out in teams, requiring strong interactions between the people involved in a project [1,13,14]. Social activity thus represents a substantial part of the daily work of a developer. Social network structures in social network sites (SNSs) emerge by adding explicit friendship connections between users. By contrast, social networks in the software engineering mainly result from object-centred sociality [15]. Developers do not just communicate with each other – they connect through shared artefacts. These social connections normally exist only within a project even though many of the artefacts used come from outside of the project. The consulted domain specific experts often do not reside within the own organisation, but in other communities.

C. *Collaborative Development Environments*

BOOCH and BROWN [2] define a CDE as "*a virtual space wherein all stakeholders of a project – even if distributed by time or distance – may negotiate, brainstorm, discuss, share knowledge, and generally labor together to carry out some task, most often to create an executable deliverable and its supporting artifacts*". So CDEs are a virtual working environment whose key functions can be clustered in the following categories: a) coordination of developers work, b) cooperation of developers, and c) formation of a community. CDEs shall create a working environment that tries to keep frictional losses at a minimum. Frictions are costs for setup and launch of the working environment, inefficient cooperation while artefact creation and dead time caused by mutual dependencies of tasks.

BOOCH and BROWN define five several stages of maturity of CDEs [2]; besides simple artefact storage (stage 1) and basic mechanisms for collaboration (stage 2), advanced artefact management (stage 3), advanced mechanisms for collaboration (stage 4) the main feature of CDEs on stage 5 is to "*encourage a vibrant community of practice*" [2].

As the current median is somewhere around stage 1 and 2 [2], it is the goal of our efforts to enhance existing CDEs for single projects with a community component that allows project-spanning collaboration. This *community-embedded CDE* (CCDE) shall provide the classical functions of a CDE stated above but also allow the seamlessly exchange of artefacts [12], data and expertise amongst projects and developers from multiple projects. The remainder of this paper describes specific requirements for a CCDE and presents an initial architectural design.

III. SOLUTION DESIGN

The following section introduces the requirements for a CCDE to support awareness and transparency in multi-project environments. We define functional and non-functional requirements for the CCDE and introduce possible data sources needed in social software engineering projects (SSEP). Finally this section provides a first architectural design of the CCDE *eCopSoft*.

A. *Organisational Requirements on a CCDE*

As stated in section 2.B, social software engineering is a collaborative development process performed by a team of people that often are separated by time and space [18]. A CCDE aims at closing the gap between the members of a team by providing project awareness and transparency as well as providing options to connect with other developers and teams. From an organisational point of view a CCDE splits into two parts: I) the developers community and II) the single projects hosted at the CCDE. The requirements for the first part of a CCDE requires methods, services and tools for networking, presentation of contents and exchange of opinions to foster data exchange and the emergence of a community feeling. Thus, a CCDE should be equipped with the typical community features of SNSs like groups, wikis, bulletin boards, user profiles and friend lists. On top of this basic services and tools the community component of a CCDE should offer domain specific areas like a job market for developers, an event review and a news corner for trending development topics. All services and tools of the developer community are to ensure the shared identity of developers, the sharing of news and opinions as well as the start of new projects.

The second important parts of a CCDE are the project spaces. A project space is basically the home of a hosted project on the CCDE. A project space has to support the members of the project in collaborative and coordinative tasks. With our CCDE we claim to foster transparency and awareness of collaborative projects, for what reason a project space must provide fundamental tools such as wikis, e-mails, repository, bug tracker, and roadmap planning. Further data sources for the deployment in software projects are discussed in section 3.B. Any user of the CCDE must be able to start a new project and easily select the required services and tools for his project. The instantiation of the single tools has to take place automatically and without human intervention. Adding new developers to a project must be possible in various ways: either the members of the project are selected a priori by the creator of the project or added to the project afterwards. For the latter one it is important to discern between public and private projects. It must be possible to allow anyone to contribute to a project (public) or to approve new developers for the project. The creator must be able to broadcast his search for new developers to the community (e.g. by sending a microblogging message or adding an entry in a bulletin board) and also to browse the existing developers in order to directly ask them to join the project.

B. *Data sources in software engineering projects*

The potential data sources relevant for software engineering project are manifold. This section tries to identify the most important resources to support collaborative software engineering in the project spaces of the CCDE.

The selection of data sources that are applicable in a CCDE is essentially dependent on the available interfaces of the respective backend systems. It is crucial that the applicable data sources provide interfaces (e.g. open APIs) that allow the installation, configuration and query of data without sweeping adaptations of the data sources. To integrate a new data source in the project spaces the implementation and upload to the server of a new connector module is sufficient.

Basically we need to distinct between data sources or systems that incorporate coordination activities and those that incorporate communication activities of the development team. The latter is to be distinguished between informal and formal communication [18]. Informal communication is considered as explicit communication via diverse communication channels such as telephone, video, audio conference, voice mail, e-mail or other verbal conversations. Formal conversation refers to explicit communication such as written specification documents, reports, protocols, status meetings or source code [6]. Thus essential systems and tools to support communication in software engineering projects include e-mail, wiki, version control systems, blogs, instant messaging or microblogs as well as shared bookmarks and shared RSS feeds. Also modern communication channels like VoIP or video chat could be part of the communicative toolbox of a project space. Coordination activities address system-level requirements, objectives, plans and issues. Working with the customer and end users carries them out. To support coordinative activities the following data sources and systems ought to be integrated in a project space: roadmap planning, issue and bug tracker, collaborative calendars, and collaborative to-do lists.

For many of the data sources mentioned well-known software systems exist that offer open APIs. Along with MediaWiki[5] and StatusNet[6], several version control systems and mail servers exist that can be a possible data source for the integration in a project space. For other data sources (e.g. shared bookmarks or VoIP) these software systems applicable in a CCDE are still to be found. Besides the open APIs it is also a necessary feature of the data sources that they store their data persistently, so that another person or tool can reuse the respective artefact in another context later.

C. *Requirements on a sophisticated Integration Layer*

The main duty of an integration layer is to process the data of all connected backend systems in a way that a central and comprehensive access to all data is possible. By integrating the different data sources into a common layer it will becomes feasible to gain additional information that could not be provided from a single backend system beforehand.

Therefore the integration layer has to be informed about changes in the different backend systems and start an analysis of the changed artefacts consequently. Changes on an artefact in a backend system have to trigger a uniform change event that can be processed and stored by the integration layer. A change event will typically deploy the analysis of the specific artefact, which requires the automatic processing of various artefact types like e-mail, wiki articles, source code and many more. Further on different analyses techniques have to be integrated pursuing different targets. These techniques ought to range from simple stuff like language detection and keyword analyses to sophisticated semantically analyses of textual artefacts and precise source code analyses. The analysis framework has to be highly extensible allowing the later addition of new techniques. All data gained throughout the analysis have to be stored in a central data structure. An efficient design of the data structure aims at fast and precise querying of the data and easy integration.

The integration layer is obliged to enhance a manually entered developer profile with automatically generated data in order to keep it up-to-date. To be able to do this and to be able to retrace the chronological sequence in the modifications of an artefact, each user interaction with one of the backend systems has to be stored as an entry in the event log of the integration layer. Additional data extracted from an event (e.g. path to a source code file, categories of a wiki entry etc.) must be stored in a global data model where artefacts are being connected system- and project spanning. With this connection it shall become possible to gain additional information about artefacts and developers and to answer specific queries like:

- Who is the main developer of a package, class or method?
- Which artefacts from other systems are highly related to the current one?
- Who is an expert in a specific development domain or technique?
- Which developers from the community could be invited to work on a new project?
- What is the expertise of a developer?

D. *Architectural design*

The requirements stated above demand for a system that allows the connection of various data sources and that

[5] http://www.mediawiki.org

[6] http://status.net/

provides multiple interfaces to access the integrated data in various ways. For that reason our prototypical implementation *eCopSoft* (event-based cooperative software engineering platform) consists of several components on different layers (cf. fig. 1) that make use of the typical mashup design pattern: easy and fast integration of multiple data sources, done by accessing APIs to produce results that were not the original reason for producing the raw source data [16].

There is a central server component (*eCopSoft* core) that is responsible for harvesting and processing data from all connected data sources on the system layer. The system layer mainly consists of the data sources described in section 3.C. From a technical point of view these systems run autonomous on a server and are connected to the *eCopSoft* server via their respective APIs. The *eCopSoft* core processes the data from all data sources, extracts event data and other metadata and stores it in an internal database. Those involved in a project can access the data stored in the backend systems and the additionally generated and aggregated metadata with various clients on the presentation layer. These tools connect to server via the *eCopSoft* API.

The *eCopSoft* application is a modular and flexible system that holds administrative and operating data, assures the connection to the backend systems and provides interfaces for accessing the operating data with various clients. Furthermore *eCopSoft* provides a central management for users and projects. The integration layer is the most important component in the *eCopSoft* architecture – all events of the backend systems are processed here. Normally an event represents a user interaction with one of the backend systems. The connector modules of the data sources act as event provider, whereas the event consumers in the integration layer process these events. Each event holds information about the user that initiated the event, the changed artefact, which kind of operation the user was carrying out (e.g. create, update, link…) as well as other event-specific information if required. On arrival of an event at the event consumers, the event and all containing information are stored in the event database. The event data is processed by the *eCopSoft* core and used to update the user profiles in the user profile database. Based on these comprehensive additional data about the usage of and work with artefacts in a development team the cooperative work can be explored in new ways. A visual project dashboard, artefact networks, artefact usage patterns or expert lists showing individual expertise are enhancing the individual and organizational learning process with artefact and user awareness and transparency.

To connect the several data sources with *eCopSoft* a connector module will be implemented for each data source. A connector module assures the creation of the project-related instances and forwards the operating data from the backend system to the integration layer. The connector modules encapsulate the specific interfaces of the backend systems represent them homogenous at server side. The creation of events can either be actively triggered by a backend system (e.g. by a SVN hook) or passively by periodically querying the data source for new data (e.g. polling a RSS feed). The automatically instantiation of the backend systems is handled via scripts as part of the *eCopSoft* application. We will script the instantiation of the backend systems because most systems do not provide an API for doing that out of the box. Furthermore a scripted instantiation allows various adaptations to meet the specific requirements of the *eCopSoft* architecture.

The clients on the presentation layer can connect to *eCopSoft* via a web services API. Mediated through the API queries for projects, developers, or artefacts are realisable. These queries can be qualified with additional criteria or weighted. Therewith it is possible to query the system for experts to a specific artefact or all artefacts that a specific developer contributed to. In the first instance we plan three main clients:

1. A web-based project home (cf. fig. 2, 3),
2. An Eclipse expert view plug-in and
3. An admin interface to administer the whole system.

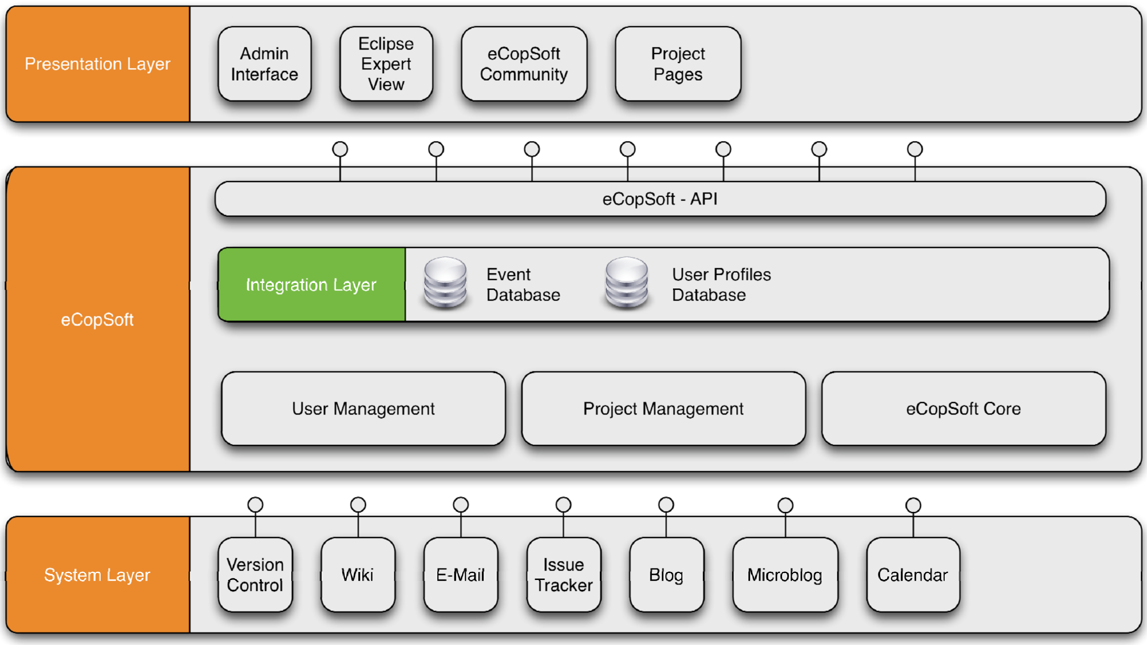

Figure 1. Schematical architecture of *eCopSoft*

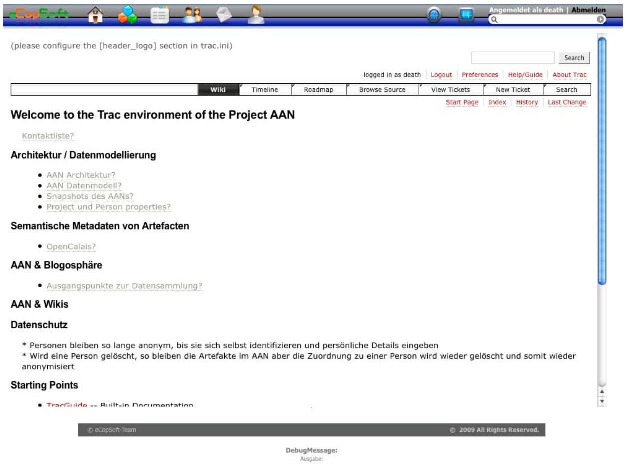

Figure 2. Screenshot of the *eCopSoft* web frontend showing a Trac environment for a project

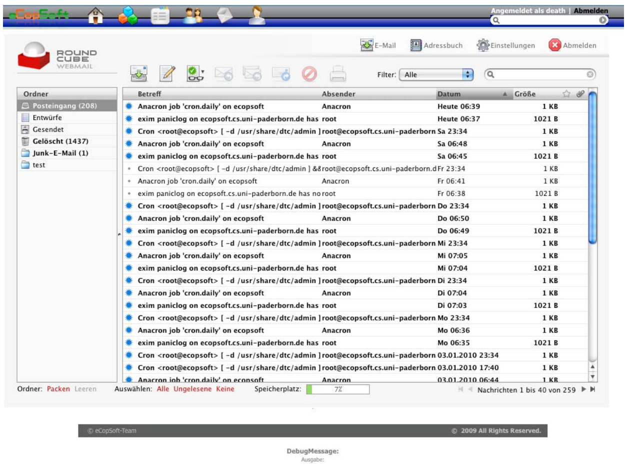

Figure 3. Screenshot of the *eCopSoft* web frontend showing the integrated webmail client for the project e-mail address

Large parts of the *eCopSoft* system base on the Java platform[7], which ensures reliability, portability and scalability. Furthermore, when it comes to problem solving, there are numerous existing Java libraries that provide finished, tested and proven solutions to specific problems. This reuse of existing frameworks accelerates the whole development process a lot. To ensure future extensibility and the integration of further connector modules, eCopSoft will be developed on an OSGi platform[8].

IV. CONCLUSION AND OUTLOOK

This paper introduced the concept of a community-embedded collaborative development environment (CCDE) whose main functions are to combine classical approaches from collaborative development environments with the strengths of communities of interest. We provided requirements on functions of a community of developers as well as functional requirements for a technical integration layer to enhance awareness and transparency in social software engineering. With the help of a sophisticated integration layer the transparency of the development process can be increased as common events connect the hitherto separated backend systems. Thereby connections between artefacts (e.g. wiki articles and Java classes) manifests that have been hidden before. On the other hand an integration layer increases the personal awareness by connecting artefacts of a project directly with its contributors and thus allowing direct communication. With the help of the automatically extended developer profile the expertise and working fields of a developer become clearer. The artefact awareness will be increased by providing related artefacts, additional metadata (semantic information, classifications, used patterns…) and a lucid overview of recent changes of artefacts. Furthermore the integration layer will allow anonymously connecting to developers from other project in order to get help from them.

Although not being a classical mashup, the presented CCDE approach connects data from various sources in a way that developers and users of the community could gain an advantage. In our opinion this advantage turns out to be in the assistance of individual work and the steady learning process by a more transparent process and enhanced awareness on various levels. Furthermore the possibility for a project spanning exchange of domain knowledge and artefacts enhances the data exchange and the collaboration within an organisation and thus fosters learning and interrelation. The easier data exchange, the higher awareness of the development process and contextualised data and experts creates an increased satisfaction with the whole development process and thus motivates developers.

The presented prototype *eCopSoft* is currently under development at the University of Paderborn and will be evaluated in software development courses. Furthermore we plan to run the CCDE as a campus-wide platform for software engineering projects, allowing the exchange of experience and data among multiple projects. The *eCopSoft* platform furthermore shall reduce the administrative overhead of providing CDEs to numerous software projects by providing a one-click-deployment for new projects. The first evaluation results of *eCopSoft* will be part of another publication.

REFERENCES

[1] N. Ahmadi, M. Jazayeri, F. Lelli, and S. Nescic, "A survey of social software engineering," in *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pp. 1–12, 2008.

[2] G. Booch, and A. W. Brown, "Collaborative development environments," in *Advances in Computers*, vol. 59, pp. 2–29, 2003.

[3] K. Beck, *Extreme Programming Explained. Embrace Change*. Addison-Wesley, 1999.

[4] J. Cross, *Informal Learning – Rediscovering the Pathways that inspire innovation and performance*. Pfeiffer, 2006.

[5] P. N. Robillard, "The role of knowledge management in software development," in *Communications of the ACM*, vol. 42, no. 1, pp. 87-94, 1999.

[6] W. Reinhardt, "Communication is the key – Support Durable Knowledge Sharing in Software Engineering by Microblogging," in *Proceedings of Conference on Software Engineering 2009, Workshop Software Engineering within Social software Environments*, 2009

[7] I. Nonaka et al., "Emergence of "Ba"," in *Knowledge Emergence*, 2001.

[8] P. Schütt, „Kleine feine Unterschiede: Daten, Information und Wissen," in *Wissensmanagement 02/2009,* pp. 10-12, 2009.

[9] A. Aurum, F. Daneshgar, J. Ward, "Investigating Knowledge Management practices in software development organisations –

[7] http://java.sun.com/

[8] http://www.osgi.org/About/Technology

An Australian experience," in *Information and Software Technology*, vol. 50, pp. 511-533, 2008.

[10] M. Polyani, *The Tacit Dimension*. Routledge & Kegan Paul, London, 1966.

[11] P. N. Robillard, and M. P. Robillard, *Types of collaborative work in software engineering*. J. Syst. Softw., vol. 53, no. 3, pp. 219–224, 2000. (doi:10.1016/S0164-1212(00)00013-3)

[12] A. Sarma, "A survey of collaborative tools in software development," Technical Report at University of Irvine, Institute for Software Research, 2005.

[13] T. DeMarco, and T. Lister, *Peopleware: productive projects and teams*. Dorset House Publishing, New York, 1987.

[14] C. Jones, *Programming productivity*. Mc Graw-Hill, New York, 1986.

[15] K. Knorr-Cetina, " Sociality with Objects: Social Relations in Postsocial Knowledge Societies," in *Theory, Culture & Society*, vol. 14, no. 4, pp. 1-30, 1997 (doi:10.1177/026327697014004001)

[16] Wikipedia. Mashup (web application hybrid). (Revision as of 10:23, 27.05.2009). Available at http://en.wikipedia.org/w/index.php?title=Mashup_(web_application_hybrid)&oldid=292635186

[17] J. Münch, and P. Liggesmyer (Eds.), *Proceedings of the Software Engineering 2009 conference, Workshops. Social Aspects in Software Engineering*, 2009.

[18] J. Herbsleb, and A. Mockus, "An empirical study of speed and communication in globally distributed software development," in *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 481–494, June 2003.

AUTHORS

Wolfgang Reinhardt is Ph.D. student at the Computer Science Education group at the University of Paderborn, Germany (e-mail: wolle@upb.de).

Sascha Rinne is doing his graduate studies at the Department of Computer Science at the University of Paderborn (e-mail: rinnes@upb.de).

This work was supported in part by the MATURE project funded by the EU under contract no. 216356 (http://mature-ip.eu).

This article was modified from a presentation at the International Conference of Interactive Computer Aided Learning ICL2009, September 2009 in Villach, Austria. Submitted 11 September 2009. Published as resubmitted by the authors on 17 January 2010.